Poul Klausen

JAVA 3

Object-oriented programming

Software Development

bookboon
LEARNING

POUL KLAUSEN

JAVA 3: OBJECT-ORIENTED PROGRAMMING

SOFTWARE DEVELOPMENT

Java 3: Object-oriented programming: Software Development
1st edition

ISBN 978-87-403-2499-0
Peer review by Ove Thomsen, EA Dania

CONTENTS

FOREWORD

This book is the third in a series of books on software development. The programming language is Java, and the language and its syntax and semantics fills obviously much, but the books have also largely focus on the process and how to develop good and robust applications. In the first book I have generally mentioned classes and interfaces, and although the book Java 2 also intesiv used classes and interfaces I have deferred the details to this book and also the next, that are dealing with object-oriented programming. It deals with how a running program consists of cooperating objects and how these objects are defined and created on the basis of the program's classes. Object-oriented programming is the knowledge of how to find and write good classes to a program, classes which helps to ensure that the result is a robust program that is easy to maintain. The subject of this book is object-oriented programming and here primarily about classes and how classes are used as the basic building blocks for developing a program. The book assumes a basic knowledge of Java corresponding to the book Java 1 of this series, but since some of the examples and exercises are relating to programs with a graphical user interface it is also assumed knowledge of the book Java 2 and how to write less GUI programs.

As the title says this series of books deals with software development, and the goal is to teach the reader how to develop applications in Java. It can be learned by reading about the subject and by studying complete sample programs, but most importantly by yourself to do it and write your own programs from scratch. Therefore, an important part of the books is exercises and problems, where the reader has to write programs that correspond to the substance being treated in the books. All books in the series is built around the same skeleton and will consist of text and examples and exercises and problems that are placed in the text where they naturally belongs. The difference between exercises and problems is that the exercises largely deals with repetitions of the substance that is presented in the text, and furthermore it is relatively accurately described what to do. Problems are in turn more loosely described, and are typically a little bigger and there is rarely any clear best solution. These are books to be read from start to finish, but the many code examples, including exercises and problems plays a central role, and it is important that the reader predict in detail studying the code to the many examples and also solves the exercises and problems or possibly just studying the recommended solutions.

All books ends with one or two larger sample programs, which focus primarily is on process and an explanation of how the program is written. On the other hand appears the code only to a limited extent – if at all – and the reader should instead study the finished program code perhaps while testing the program. In addition to show the development of programs that are larger than the examples, which otherwise is presented, the aim of the concluding examples also is to show program examples from varying fields of application.

Most books also ends with an appendix dealing with a subject that would not be treated in the books. It may be issues on the installation of software or other topics in computer technology, which are not about software development, but where it is necessary to have an introductory knowledge. If the reader already is familiar with the subject, the current appendix can be skipped.

The programming language is, as mentioned Java, and besides the books use the following products:

- NetBeans as IDE for application development
- MySQL to the extent there is a need for a database server (from the book Java 6 onwards)
- GlassFish as a web server and application server (from the book Java 11 onwards)

It is products that are free of charge and free to install, and there is even talk about products, where the installation is progressing all by itself and without major efforts and challenges. In addition, there are on the web detailed installation instructions for all the three products. The products are available on Windows and Linux, and it therefore plays no special role if you use Linux or Windows.

All sample programs are developed and tested on machines running Linux. In fact, it plays no major role, as both Java and other products work in exactly the same way whether the platform is one or the other. Some places will be in the books where you could see that the platform is Linux, and this applies primarily commands that concerning the file system. Otherwise it has no meaning to the reader that the programs are developed on a Linux machine, and they can immediately also run under Windows unless a program refers to the file system where it may be necessary to change the name of a file.

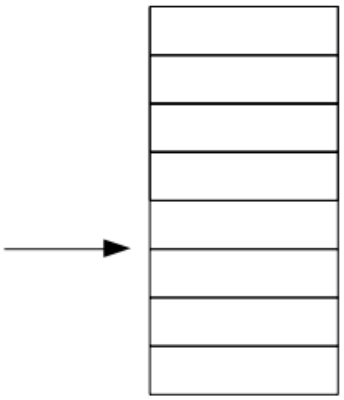
Finally a little about what the books are not. It is not "how to write" or for that matter reference manuals in Java, but it is as the title says books on software development. It is my hope that the reader when reading the books and through the many examples can find inspiration for how to write good programs, but also can be used as a source collection with a number of examples of solutions to concrete everyday programming problems that you regularly face as a software developer.

1 INTRODUCTION

Programs should deal with data, and data must be represented and stored, which happens in variables. A language like Java has a number of built-in types to variables (as explained in the book Java 1), but often you need to define your own data types that better reflects the task the program must solve. This is where the concept of a class comes in. A class is something that define a concept or thing within the problem area, and that can be said a lot about what a classes is, and what is not, but technically a class is a type. It's a bit more than just a type, as a class partly defines how data should be represented, but also what we can do with the data of that type. A class defines both how the type will be represented, and what operations you can perform on variables of that type. Classes are also generally been described in the book Java 1, but in this book I will start or continue with a more detailed discussion of Java's class concept.

When wee create variables whose type is a class, wee talk about objects, so that a variable of a class type is called an *object*, but really there is no great difference between a variable and an object, and there is well along the road no reason to distinguish between the two. But dig a little further down, there is a reason that has to do with how variables and objects are allocated in the machine's memory.

Variables whose type is *int*, *double*, *boolean*, *char*, etc., are called *simple variables* or *primitive types*. To a running program is allocated a so-called *stack* that is a memory area used by the application among other to store variables. The stack is highly efficient, so that it is very fast for the program to continuously create and remove variables as needed. This is called a stack, because one can think of the stack as a data structure illustrated as follows:



When the program creates a new variable, it happens at the top of the stack – where the arrow is pointing, and if a variable must be removed, it is the one that lies at the top of the stack. Variables of simple data types have the property that they always take up the same. As an example fills an *int* the same (4 bytes), no matter what value it has. Therefore for the kind of variables stored directly on the stack the compiler knows how much space they use. Thus, if one as an example in a method writes

```
int a = 23;
```

then there will be created a variable on the stack of type *int* and with the space as an *int* requires (4 bytes), and the value is stored there. Variables that in this way are stored directly on the stack, are also called *value types*, and the simple or primitive types – except *String* – are all value types.

Things are different with the variables of reference types such as variables, that has a class type. They have to be created explicitly with *new*. If, for example you have a class named *Cube*, and you want to create such an object, you must write

```
Cube c = new Cube();
```

It looks like, how to create a simple variable. The variable is called *c*. When *new* is executed, what happens is that on the a so-called heap is created an object of the type *Cube*. One can think of the heap as a memory pool from which to allocate memory as needed. On the stack is created an usual variable of type *Cube*, but stored on the stack is not the value of a *Cube* object, but rather a reference to the object on the heap. All references take up the same regardless of the type, and they can then be stored on the stack. That is why it is called a *reference type*. Where exactly an object is created on the heap is determined by a so-called heap manager that is a program that is constantly running and manages the heap. It is also the heap manager, which automatically removes an object when it is no longer needed.

For the above reasons, it is clear that the variables of value types are more effective than the objects of reference types. It in no way means that objects of reference types are ineffective, and in most cases it is not a difference that you need to relate to, but conversely there are also situations where the difference matters. Thus, it is important to know that there are big differences in how value types and reference types are handled by the system, and that in some contexts it are of great importance for how the program will behave, but there'll be many examples that clarify the difference. So far, it is enough to know that the data can be grouped into two categories depending on their data type, so that the data of value types are allocated on the stack and is usually called variables, while data of reference types are allocated on the heap and called objects – although there is no complete consistency between the two names.

In the bokk Java 1, I defined the following class, which represents a usual cube:

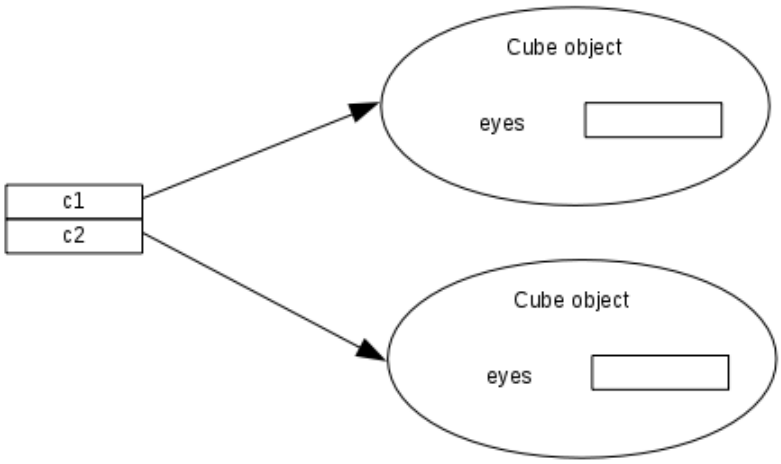
```
public class Cube
{
 private static Random rand = new Random();
 private int eyes;

 public Cube()
 {
  roll();
 }

 public int getEyes()
 {
  return eyes;
 }

 public void roll()
 {
  eyes = rand.nextInt(6) + 1;
 }

 public String toString()
 {
  return "" + eyes;
 }
}
```

The value of the *Cube* or it's state is represented by an *int*, while its properties or behavior is represented by three methods. In a program, you can then create objects of type *Cube* as follows:

```
Cube c1 = new Cube();
Cube c2 = new Cube();
```

When you create an object of a class, the class's instance variables are created, and then class's constructor is executed. If a class has no constructor, there automatically will be created a default constructor – a constructor with no parameters. A constructor is characterized in that it is a method which has the same name as the class and do not have any type. A constructor is a method, but it can not be called explicitly and is performed only when an object is instantiated. The class *Cube* has a constructor, that is a default constructor.

The objects are as mentioned not allocated on the stack, but is created on the heap. *c1* and *c2* are usual variables on the stack, but does not include the objects but include instead references to the two objects on the heap. It's rare you as a programmer need to think about it, but in some situations it is important to know the difference between an object allocated on the stack and on the heap. The figure below illustrate how it looks in the machine's memory with the two variables on the stack that refer to objects on the heap:



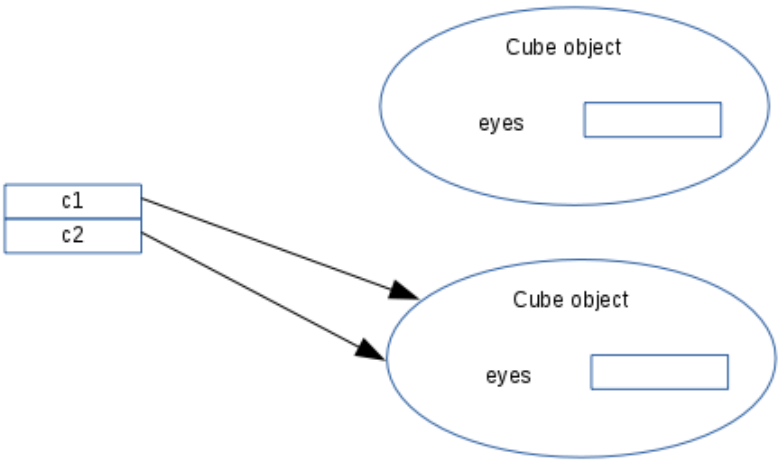
If, for example you in the program writes

```
c1 = c2;
```

it means that there no longer is a reference to the object as *c1* before referring to, but there are instead two references to the object as *c2* refers to (see the figure below). If, you then writes

```
c1.roll();
c2.roll();
```

it means that the same cube is rolled twice because both references refer to the same object. When there are no longer are any references to an object, it also means that the object is automatically removed from heap by the heap manager and the memory that the object used, will be deallocated.



A class is referred to as a type, but it is also a concept of design. The class defines objects state as instance variables, how objects are created, and which memory is allocated for the objects. Objects have at a given moment a value as the content of the instance variables, and an object's value is referred to as its state. The class also defines in terms of its methods, what can be done with the object, and thus how to read and modify the object's state. The methods define the object's behavior.

Every Java program consists of a family of classes that together defines how the program would operate and a running program will at a given time consist of a number of objects instantiated from the program's classes, objects that work together to accomplish what the program now must do. The work to write a program is then to write the classes that the program should consist of. Which classes that is, are in turn, not very clearly, and the same program may typically be written in many ways made up of classes, which are quite different. The work to determine which classes a program should consist of and how these classes should look like in terms of variables and methods is called design. In principle, one can say that if a program does the job properly, it may be irrelevant, which classes it is composed of, but unsuitable classes means

- that it becomes difficult to understand the program and thus to find and fix errors
- that it in the future will be difficult to maintain and modify the program
- that it becomes difficult to reuse the program's classes in other contexts

Therefore, the design and choice of classes, is a very key issue in programming, and in that context wee are speaking of program quality (or lack thereof). A class is more than just a type, but it is a definition or description of a concept from the program's problem area. When you have to define which classes a program must consist of, you must therefore largely focus on classes as a tool to define important concepts more than the classes as a type in a programming language.

An object is characterized by four things:

- a unique identifier that identifies a particular object from all other
- a state that is the object's current value
- a behavior that tells what you can do with the object
- a life cycle from when the object is created and to it again is deleted

An object is created from a class, and is that time assigned a reference that identifies the object. The class's instance variables determine which variables to be created for the object, and the value of these variables is the state of the object. The class's methods define what you can do with the object and thus the object's behavior. The last point of concerning is the objects life cycle, which is explained later.

To program in that way in which you see a program as composed of a number of cooperating objects, is called *object-oriented programming*. For many years wee have used object-oriented programming, and although it is primarily a question of design, it also requires that the current programming language supports the object-oriented thinking, and this is what makes Java an object oriented programming language. This book focuses on object-oriented programming, including both object-oriented design, and how it is supported in Java. There are many concepts that includes

1. objects life cycle
2. inheritance and polymorphism
3. generic types
4. collection classes

all subjects that are discussed below, and in the next book. To explain these concepts I will consider classes as concerning students at an educational institution, as well as books in a library.

2 CLASSES

In the book Java 1, I have generally described classes, but there is much more to say, and classes are the whole fundamental concept in object-oriented programming. This book is therefore about how to write classes and how, based on these classes to create objects. I'll start with a class that can represent a course at an educational institution:

```
package students;

/**
 * Class which represents a subject associated with an education.
 * A subject consists in this context an id, which is an acronym that
 * identifies the subject and a name.
 */
public class Subject
{
 private String id;     // the subject id
 private String name;   // the subject's name
```

```
/**
 * Creates a new subject. A subject must have both an ID and a name, and is it
 * not the case, the constructor raises an exception.
 * @param id The subjects id
 * @param navn The subjects name
 * @throws Exception If the id or the name are illegal
 */
public Subject(String id, String name) throws Exception
{
 if (!subjectOk(id, name))
  throw new Exception("The subject must have both an ID and a name");
 this.id = id;
 this.name = name;
}

/**
 * @return The subjects id
 */
public String getId()
{
 return id;
}

/**
 * @return The subjects name
 */
public String getName()
{
 return name;
}

/**
 * Changes the subject's name. The subject must have a name, and if the parameter
 * does not specify a name the method raises a exception.
 * @param navn The subjects name
 * @throws Exception If the name is null or blank
 */
public void setName(String name) throws Exception
{
 if (!subjectOk(id, name)) throw new Exception("The subject must have a name");
 this.name = name;
}

/**
 * @return The subjects name
 */
```

```
 public String toString()
 {
  return name;
 }' /**

  * Method tthat tests whether two strings can represent a subject.
  * @param id The subjects id
  * @param name The subjects name
  * @return true, if id and name represents a legal subject
  */
 public static boolean subjectOk(String id, String name)
 {
  return id != null && id.length() > 0 && name != null && name.length() > 0;
 }
}
```

Compared to what I have said about classes in the book Java 1 there is not much new to explain, and the meaning of variables and methods are explained in the class as comments, but you must be aware of the following.

Classes are defined by the reserved word *class*, and a class has a name as here is *Subject*. In Java, it is recommended that you write the name of classes capitalized, but otherwise applies the same rules for classes names, that makes the names of variables. A class can have a visibility that may be *public* or *private*, but so far I will define all classes *public*. Visiblity says something about who may use the class.

Classes consist of variables and methods, and there is in principle no upper limit for any of the two parts. Both variables and methods can have *public* and *private* visibility. If a member (variable or method) is *public*, it can be referenced from methods in all other classes, that have an object of the current class, and if it is *private*, it can only be referenced by methods in the class itself. Usually you define variables as *private*, while the methods to be used by other classes, are defined *public*. Put a little different then a class defines with *public* methods, what can be done with objects of the class and thus the objects behavior.

In this case, the class has two variables, both of which are of the type *String*. Such variables are called *instance variables*, and each object of the class has its own copies of these variables whose values are the object's *state*. The type *String* is also a class type, and the two variables are therefore reference types. They do not refer necessarily to anything and such variables has the value *null* (*null* reference), which simply means that you has reference variables, which does not refer to an object.

Classes can have one or more constructors that are special methods executed when creating an object of the class. A construction is written in the same way as other methods, but the name is the same as the class, and they do not have any type. Constructors can in principle perform anything, but they are typically used to initialize instance variables in the class. In this case, the constructor has parameters for an object's values. These parameters have the same names as the instance variables (it is not a requirement, and the parameters must be named anything), and in the constructor's code there are two things with the same name (both an instance variable and a parameter). It is solved by the word *this* where, for example *this.name* refer to the instance variable, while *name* specific the parameter. I will insist that a specific subject must have both an id and a name. Therefore, the constructor tests the parameters, and if they not both how a value the constructor raises an exception. Exceptions are discussed later, but if a method as here can raise an exception, the method must be marked with *throws*:

```
throws Exception
```

If the parameters do not have legal values, raises the constructor an exception

```
if (!subjectOk(id, name))
 throw new Exception("The subject must have both an ID and a name");
```

You should note that the parameters are tested by the method *subjectOk()*. This method is defined *static*, meaning that it can be used without having created an object of the class. Other classes can use the method to test values before creating the object. That kind of control methods may be reasonable, if you later find out that the controls should be different. You should then simply change the method, and it is not necessary to change elsewhere in the code.

That a constructor in this way raises an exception is a guarantee that no one instantiates illegal objects – the one that instantiates an object, must necessarily decide what should happen in case the object is illegal. However, we can discuss the wisdom of in this way to let constructor validate input parameters and possible raise an exception, and the diskusion I will return to later.

In this case, the class's other methods are simple. An object's id must be readable, but it must not be changed. Therefore, the class has only a method

```
public String getId()
```

to returns the value. Methods which in that way just returns the value of an instance variable, usually all use this syntax (the variable's name with the first character in uppercase and prefixed with the word *get*), and you call them for *get*-methods. Similarly, the variable *name* also has a get method, but here it must also be possible to change the name. The class then has a *set*-method for *name*:

```
public void setName(String name)
```

and methods which in that way only are used to change the value of an instance variable, usually all use this syntax. In this case, the method may raise an exception if the name has an illegal value, but it is not a requirement that the *set*-methods may raise an exception.

Both the class and its methods are decorated with Java comments. With a special tool you can from these comments create a full finished html documentation of classes in a program. I'll explain how later, and also what the various descriptions elements means, but in this case, it is easy enough to figure out the meaning. If you place the cursor in front of a method and enters

```
/**
```

plus the Enter key, then NetBeans creates a skeleton for a comment, and one of the advantages is, that in that way you remember to write comments for a method's parameters and return value and any exceptions.

Objects must be created or instantiated, which is done with *new*:

```
package students;

public class Students
{
 public static void main(String[] args)
 {
  try
  {
   Subject subject = new Subject("MAT7", "Matematichs");
   System.out.println(subject);
   subject.setName("Matematichs 7");
   System.out.println(subject);
  }
  catch (Exception ex)
  {
   System.out.println(ex.getMessage());
  }
 }
}
```

The program creates an object *subject* of the type *Subject*. When the constructor in the class *Subject* may raise an exception, it should be handled, and it is necessary to place the code that may raise an exception in a try/catch. After the object is created, it is printed on the screen. If you prints an object with *println()*, it is the value of the object's *toString()* method that is printed. In principle, all classes has a *toString()* method which returns a text that represents a particular object. Finally the program changes the name of the subject, and the object is printed again.

The classes (*Subject* and *Students*) described above can be found in the project *Students1*.

EXERCISE 1

Create a new project in Netbeans that you can call *Library*. You must add a class that should be called *Publisher* that represents a book publisher when the publisher must has the following two properties:

1. an integer, that you can call *code* which is to be interpreted as a publisher identifier
2. a name, that just is a text and is the publishers name

It is a requirement that the publisher number is a positive integer and that a publisher must have a name. The class must have a constructor that has parameters for both variables and has the following *public* methods:

- *get* methods to both variables
- a *set* method to the publishers name
- a *toString()* method that returns the publishers name followed by the publisher number in square brackets

When you have written the class, you must document it and its methods using Java comments. Finally, in the *main* class – that class *Library* – you should write a *main()* method that

- Creates a *Publisher* object – you decide the values
- Prints the publisher
- Change the publishers name
- Prints the publisher again

If you executes the program, the result could be the following:

```
The new Publisher [123]
The old Publisher [123]
```

2.1 MORE CLASSES

I will then look at a class representing a subject that a student has completed or participate in. It is assumed for simplicity that the same subjects only can be performed once a year and therefore are identified the subject's id and the year. The project is called *Students2* and is created as a copy of *Strudents1*. There is added the class *Course* as shown below, and again explains the comments of the individual methods there purposes:

```
package students;
/**
 * Class that represents a course, where a course regarding a year and a subject.
 * A student may have completed or attend a particular course.
 * It is an assumption that the same subjects only be executed once a year.
 * A course can also be associated with a character. If so, it means that
 * the student has completed the course.
 */
public class Course
{
 // year of when the course is completed or offered
 private int year;

 // the subject that the course deals
 private Subject subject;

 // the character tkhat a student has obtained in the subject
 private int score = Integer.MIN_VALUE;

 /**
  * Creates a course for a concrete subjects and a given year.
  * @param year The year for the course
  * @param subject The subject for this course
  * @throws Exception If the year is illegal or the subject is null
  */
 public Course(int year, Subject subject) throws Exception
 {
  if (!courseOk(year, subject)) throw new Exception("Illegal course");
  this.year = year;
  this.subject = subject;
 }

 /**
  * A cource is identified by the subjects id and the year
  * @return ID composed of the year of the subject's id separated by a hyphen
  */
 public String getId()
 {
```

```
 return year + "-" + subject.getId();
}
/**
 * @return The year where the course is held.
 */
public int getYear()
{
 return year;
}
/**
 * @return true, if the student has completed the course
 */
public boolean completed()
{
 return score > Integer.MIN_VALUE;
}

/**
 * @return The character that the student has achieved
 * @throws Exception If a student has not obtained a character
 */
public int getScore() throws Exception
{
 if (score == Integer.MIN_VALUE)
  throw new Exception("The student has not completed the course");
 return score;
}

/**
 * Assigns this course a score.
 * @param score The score that is the obtained
 * @throws Exception If the score is illegal
 */
public void setScore(int score) throws Exception
{
 if (!scoreOk(score)) throw new Exception("Illegal ckaracter");
 this.score = score;
}

/**
 * Assigns this course a character.
 * @param score The score that is the obtained
 * @throws Exception If the score is illegal
 */
public void setScore(String score) throws Exception
{
 try
 {
```

```
   int number = Integer.parseInt(score);
   if (!scoreOk(number)) throw new Exception("Illegal score");
   this.score = number;
  }
  catch (Exception ex)
  {
   throw new Exception("Illegal score");
  }
}

/**
 * @return The course's subject
 */
public String toString()
{
 return subject.toString();
}

/**
 * Tests whether the parameters for a course are legal
 * @param year The year for the course
 * @param subject The subject that this course deals
 * @return true, If the year is illegal or the subject is null
```

```
   */
  public static boolean courseOk(int year, Subject subject)
  {
   return year >= 2000 && year < 2100 && subject != null;
  }

  // Validates whether a character is legal
  private boolean scoreOk(int score)
  {
   return true;
  }
}
```

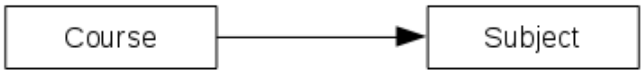
The class looks like the class *Subject*, but there are still things that you should note. The class has three instance variables, and one of the type *Subject*. You should therefore note that the type of an instance variable may well be a user-defined type and here a class. There is also an instance variable called *year* that has the type *int*. There is a variable *score*, that should be applied to the score which a student has achieved in the subject. In one way or another it should be possible to indicate that a student has not yet been given a score, for example because the course has not been completed. This is solved by assigning the variable the value *Integer.MIN_VALUE* which is the least 32-bit integer that can occur. This number is -2147483648, and one must assume that this score can not occur in a character system.

Note that wee in Java uses the same rules and conventions for method names that wee use for variables and such should the name of a method to start with a lowercase letter.

The method *setScore()*, which assigns a course a character is available in two versions. In Java, a method is identified by its name and its parameters. There may well in the same class be two methods with the same name as long as they have different parameters, that is when the parameter lists can be separated either on the parameter types or the number of parameters. It is sometimes said that the methods can be overloaded. In this case, as mentioned, there are two versions of the method *setScore()*, one with an *int* as a parameter, while the other has a *String*. So you can specify a character as both an *int* and a *String*, and the possibility is included only to illustrate the concept of function overloading.

When you assign a course a score it is validated by the method *scoreOk()*. This method is trivial, since it always returns true, and it provides little sense. The goal is that the method at a later time should be changed to perform something interesting, and you can say that the problem is that the class *Course* not have knowledge of what is legal scores. There are several grading systems.

Now I have defined two classes regarding students, and there is a connection between the two classes so that the class *Course* consist of an object of the class *Subject*. It is sometimes illustrated as



If you have to create a *Course*, you must provide a year and a *Subject*, which is an object that must also be created. If you find it appropriate, you can add an additional constructor that creates this object:

```
/**
 * Creates a course for a concrete subjects and a given year.
 * @param year The year for the course
 * @param id The subject's id
 * @param name The subject's name
 * @throws Exception If the year is illegal or id and name are not legal
 */
public Course(int year, String id, String name) throws Exception
{
 subject = new Subject(id, name);
 if (!courseOk(year, subject)) throw new Exception("Illegal year");
 this.year = year;
}
```

You must specifically note that if the class *Subject* raises an exception (the constructor of the class *Subject*) when the object is created, then the above constructor is interrupted by sending the exception object from the class *Subject* on to the calling code. Constructors can thus be overloaded by the exact same rules as for other methods.

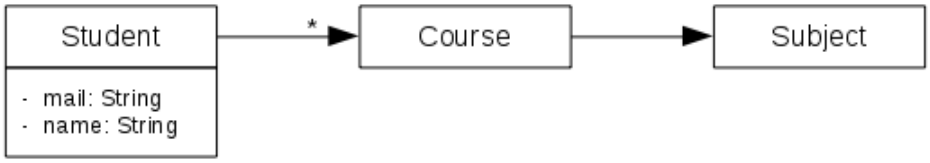
Below is a method that creates two courses:

```
private static void test1()
{
 try
 {
  Course course1 = new Course(2015, new Subject("MAT6", "Matematik 6"));
  Course course2 = new Course(2015, "MAT7", "Matematik 7");
  course1.setScore(7);
  print(course1);
  print(course2);
 }
 catch (Exception ex)
```

```
  {
   System.out.println(ex.getMessage());
  }
}

private static void print(Course course) throws Exception
{
 System.out.println(course);
 if (course.completed())
  System.out.println("The course is completed with the result "
   + course.getScore());
}
```

Note how to create objects of the type *Course* and how the two different constructors are used. You should note that the method *getScore()* may raise an exception. Therefore, the *print()* method may raise an exception, and if it happens the method will be interrupted and the *Exception* object is passed on to the method *test1()*. Also note how in the *print()* method you refers to methods in the *Course* class using the dot operator on the *course* object, and that the methods works on the specific *Course* object's state.

I will then look at a class that represents a student. A student must in this example alone have a mail address and a name, and also a student could have a number of courses that the student has either completed or participates in. This can be illustrated as follows:



where * indicates that a student may be associated with many *Course* objects. The class is shown below, and it takes up a lot particular because of the comments, but to emphasize that classes should be documented by comments, I have chosen to retain them, and they also helps to explain the meaning of class's methods:

```
package students;

import java.util.*;
/**
 * The class represents a student when a student is characterized by a mail
 * address.
 * As a unique identifier is used, a number that is automatically incremented
 * by 1 each time a new object is created.
 * A student has a list of courses that the student has completed or
 * participate in.
 */
public class Student
{
 private static int nummer = 0;                // the last used id
 private int id;                               // the students id
 private String mail;                          // the student's mailadresse
 private String name;                          // the student's name
 private ArrayList<Course> courses = new ArrayList();  // a list with courses

 /**
  * Creates a new student from an array of courses.
  * @param mail The student's mail address
  * @param name The student's name
  * @param course An array with courses
  * @throws Exception If the mail or the name does not represents legal values
  */
 public Student(String mail, String name, Course … course) throws Exception
 {
  if (!studentOk(mail, name))
   throw new Exception("Ulovlig værdier for en studerende");
```

```
 this.mail = mail;
 this.name = name;
 for (Course c : course) courses.add(c);
 id = ++nummer;
}
static
{
 nummer = 1000;
}

/**
 * @return The student's id
 */
public int getId()
{
 return id;
}

/**
 * @return The students mail address
 */
public String getMail()
{
 return mail;
}

/**
 * Changes the mail address on a student.
 * @param mail The student's mail address
 * @throws Exception If no legal mail address is specified
 */
public void setMail(String mail) throws Exception
{
 if (mail == null || mail.length() == 0)
  throw new Exception("Illegal mail adresse");
 this.mail = mail;
}

/**
 * @return The student's name
 */
public String getName()
{
 return name;
}
```

```
/**
 * Changing the name of a student.
 * @param name The student's name
 * @throws Exception If no name is specified
 */
public void setName(String name) throws Exception
{
 if (name == null || name.length() == 0) throw new Exception("Illigal name");
 this.name = name;
}

 /**
 * @return Number of courses that this student has completed or participated in
 */
public int getCount()
{
 return courses.size();
}

/**
 * Returns the n'th course for this student
 * @param n Index of the desired course
 * @return The nth course like this student has completed or participated in
```

```
 * @throws Exception If the is illegal
 */
public Course getCourse(int n) throws Exception
{
 return courses.get(n);
}

 /**
 * @param id Identifier of a course
 * @return Teh course identified by id
 * @throws Exception If the course is not found
 */
public Course getKursus(String id) throws Exception
{
 for (Course c : courses) if (c.getId().equals(id)) return c;
 throw new Exception("Course not found");
}

 /**
 * Returns the courses that the current student have completed or participated
 * in a particular year.
 * @param year The year searching for
 * @return Course list of discovered courses
 */
public ArrayList<Course> getCourses(int year)
{
 ArrayList<Course> list = new ArrayList();
 for (Course c : courses) if (c.getYear() == year) list.add(c);
 return list;
}

 /**
 * Adds a course for the students.
 * @param course The course to be added
 * @throws Exception If the list already have the same course
 */
public void add(Course course) throws Exception
{
 for (Course c : courses)
  if (course.getId().equals(c.getId()))
   throw new Exception("The course is already added");
 courses.add(course);
}

/**
 * @return Returns the student's id address and name
```

```
 */
public String toString()
{
 return "[" + id + "] " + name;
}

 /**
 * Tests whether the mail and name to represent a student.
 * @param mail A student's mail address
 * @param name A student's name
 * @return true, if mail and name represents a legal student
 */
public static boolean studentOk(String mail, String name)
{
 // the control is simple and should be extended to check whether mail has a
 // proper format for an email address
 return mail != null && mail.length() > 0 && name != null && name.length() > 0;
}
```

The class *Student* represents a student. The class has four instance variables, where the first is an identifier (a number), the next two are of the type *String* and is respectively the mail address and the student's name. The last is of the type *ArrayList<Course>*, and should be used to the courses that the student has completed or participates in. The class's constructor raises an exception if the parameters are not legal. The class's other methods require no special explanation, but note, however, that the method *getCourses()* is overloaded.

An object has a variable *id*, which is a number that can identify a student. This number is automatically assigned in the constructor starting with 1001 and such that the next student is assigned the next number. How it exactly works, I will return to when I talk about static members.

Below is a method, which creates two students:

```
private static void test2()
{
 try
 {
  Student stud1 = new Student("svend@mail.dk", "Svend Andersen");
  Student stud2 = new Student("gorm@mail.dk", "Gorm Madsen",
   new Course(2015, new Subject("PRG", "Programming")),
   new Course(2015, new Subject("OPS", "Operating systems")));
  stud1.add(new Course(2014, new Subject("DBS", "Database Systems")));
  stud2.add(new Course(2014, new Subject("WEB", "Web applicattions")));
```

```
   stud1.getCourses("2014-DBS").setScore(4);
   stud2.getCourses("2014-WEB").setScore(10);
   stud2.getCourses("2015-OPS").setScore(2);
   print(stud1);
   print(stud2);
  }
  catch (Exception ex)
  {
   System.out.println(ex.getMessage());
  }
}

private static void print(Student stud)
{
 System.out.println(stud);
 for (int i = 0; i < stud.getCount(); ++i)
 {
  try
  {
   Course c = stud.getCourse(i);
   System.out.println(c + ", " +
     (c.completed() ? "Score: " + c.getScore() : "Not completed"));
  }
```

```
    catch (Exception ex)
    {
    }
  }
}
```

The first student is created without specifying courses. The other student is created with two courses. Next, the program adds a course to both students and finally assigns scores for three courses, and the two students are printed. The method is executed, the result is:

```
[1001] Svend Andersen
Database Systems, Score: 4
[1002] Gorm Madsen
Programming, Not completed
Operating systems, Score: 2
Web applicattions, Score: 10
```

EXERCISE 2

This exercise is a continuation of exercise 1. Start by making a copy of the project *Library* as you can call *Library1* and open the copy in NetBeans. You must add a class named *Author*. The class should have three instance variables

1. *id* which is an integer, that can identify an author
2. *firstname* for the author's first name
3. *lastname* for the author's last name

It is a requirement that the first name is not *null*, but it is allowed to be blank. On the other hand, an author must have a last name.

The class should have a constructor, which has two parameters, respectively to initialize *firstname* and *lastname*. A author's id should be assigned automatically, so that every time a new *Author* is created the object get an id, which is one greater than the previous one. You can solve this a static variable:

```
private static int counter = 0;
```

which is counted up by 1 each time a new *Author* is created.

Regarding the methods, the class must have a *get* method for all the three variables, and a *set* method for both *firstname* and the *lastname*. Finally, there must be a *toString()* method that returns an author's first and last name separated by a space.

Once you have written the class, you must document it using Java comments.

In the main class, write a method that can print an *Author*:

```
private static void print(Author a)
```

when the method should print the author's id listed in brackets, as well as the author's first and last name. Write the *main()* method so it performs the following:

- creates two authors – you decides the names
- prints the two authors
- change the first name of the first author
- change the last name of the other author
- prints the two authors again

If you executes the program the result could be:

```
[1] Gudrun Jensen
[2] Karlo Andersen
[1] Abelone Jensen
[2] Karlo Kristensen
```

EXERCISE 3

Make a copy of the project *Library1* and call the copy *Library2*. Open it in NetBeans. Add a class *Book* that represents a book with the following characteristics:

- Isbn, that must be declared
- Title, that must be declared
- Publisher year, there must be 0 or lie between 1900 and 2100, 0 indicates that publisher year is unknown
- Edition, which must be between 0 and 15 (inclusive), 0 indicates that the edition is unknown
- Number of pages that must be non-negative, 0 indicates that number of pages are unknown
- Publisher, there must be a *Publisher* object or *null*, where *null* means that the publisher is unknown
- A number of authors, there must *Author* objects (the number af authors must be 0 if no authors are known)

The class must have four constructors with the following signatures:

```
public Book(String isbn, String title) throws Exception
{
 …
}

public Book(String isbn, String title, int released) throws Exception
{
 …
}

public Book(String isbn, String title, Publisher publisher) throws Exception
{
 …
}

public Book(String isbn, String title, int
released, int edition, int pages,
  Publisher publisher) throws Exception
{
 …
}
```

There must be *get* methods for all of the above fields, and there must be *set* methods to the fields *title*, *released*, *edition*, *pages* and *publisher*.

As for authors the class should in the same manner as the class *Student* have an *ArrayList<Author>* the *Author* objects:

```
private ArrayList<Author> authors = new ArrayList();
```

The class *Book* should have a *toString()* method must return the book's ISBN and the *title* separated by a space. Finally, there must be a static method:

```
public static boolean isbnOk(String isbn)
```

that validates an isbn, when it so far only should test that the parameter is not an empty string.

Once you have written the class, you must document it and its methods.

In the main class, write a method that can create and return a *Book* object from information on the book's properties:

```
private static Book create(String isbn, String
title, int released, int edition,
  int pages, Publisher publisher, Author … authors) throws Exception
{
}
```

You must then write a method that can print a book:

```
private static void print(Book book) {}
```

when it has to print

- the book's ISBN
- the book's title
- the book's publisher year if known
- the book's edition if known
- the book's number of pages if known
- the book's publisher if known
- the book's authors if there are authors

You must finally write the *main()* method, so it performs the following:

- create a book using the above method where you specify values for all the book's fields and including the book's author(s)
- create a book where you only initialize the book's ISBN and title
- prints the two books
- change the publisher year, edition, number of pages and the publisher for the last book and adds the author(s)
- prints the last book again

The result could, for instance be as shown below:

```
ISBN:        978-1-59059-855-9
Title:       Beginning Fedora From Noice to Professional
Released:    2007
Edition:     1
Pages        519
Publisher:   The new Publisher [123]
Authors:
[1] Shashank Sharma
[2] Keir Thomas

ISBN:        978-87-400-1676-5
Title:       Spansk Vin
ISBN:        978-87-400-1676-5

Title:       Spansk Vin
Released:    2014
Edition:     1
Pages        335
Publisher:   Politikkens Forlag [200]
Authors:
[3] Thomas Rydberg
```

PROBLEM 1

A book is characterized by an ISBN, which is an international numbering on 10 or 13 digits. The system was introduced around 1970, and until 2007 it was on 10 digits, which was divided into four groups separated by hyphens:

```
99-999-9999-9
```

wherein the last group always consists of a single digit (character) and is a control character. The other three groups are interpreted as follows:

1. the first group is a country code
2. the second group is the publisher identifier
3. the third group is the title number

The control character is calculated by the modulus 11 method, using the weights 10, 9, 8, 7, 6, 5, 4, 3, 2 and 1. This is best explained with an example. Consider a concrete ISBN:

```
1-59059-855-5
```

To determine the check character you determine the following weighted sum:

```
 1    5    9    0    5    9    8    5    5
10    9    8    7    6    5    4    3    2
-------------------------------------------

10 + 45 + 72 + 0 + 30 + 45 + 32 + 15 + 10 = 259
```

You now determines the rest by dividing by 11 and you get

```
259 % 11 = 6
```

If this rest is 0, the check digit is 0. If the remainder is 10, X is the check charracter, and otherwise, one uses 11 minus the remainder as check digit. As the rest this time is 6, you get check 11 – 6 = 5.

Eventually it was realized that by the years there vould be too few ISBN numbers, and therefore it was decided to change the system from 2007. From that time the numbers are preceded by a 3-digit prefix, and the numbers thus came to consist of another group, that always has 3 digits:

```
999-99-999-9999-9
```

So far only of the numbers 978 and 979 are used as prefixes, but in the future there will probably be other opportunities. In addition to increasing the numbers with this prefix the calculation of the control character was also changed, so that as weights using alternating values of 1 and 3, and the control character is then the modulus 10 of the weighted sum. Again, it is easiest to illustere the calculations with an example. Consider

```
978-1-59059-855-9
```

You calculate the following weighted sum:

```
  9    7    8    1    5    9    0    5    9    8    5    5
  1    3    1    3    1    3    1    3    1    3    1    3
-------------------------------------------------------------
  9 + 21 + 8 + 3 + 5 + 27 + 0 + 15 + 9 + 24 + 5 + 15 = 141
```

Then you calculate rest at division by 10:

```
141 % 10 = 1
```

If this remainder is 0, it is control character. Or is the check digit 10 minus the remainder, and in this case, it is 10 – 1 = 9.

You must now return to the project *Library* and the class *Book*. Start by creating a copy of the project *Library2* and call the copy *Library3*. Open the copy in NetBeans. The class *Book* has a static method *isbnOk()* to validate whether a string is a legal ISBN. The control is trivial, but you must now change the code to validate an ISBN folowing the above rules.

You must also write a test method that tests whether the control method validates correctly. It is important that you test with multiple numbers, so you get all the cases, and it is important that you also test illegal numbers.

2.2 METHODS

In the explanation of classes I have already dealt with methods, but there are a few concepts that you should be aware of. As mentioned, a method is identified by its name and the parameter list. The parameters that you specify in the method definition, are called the *formal parameters* and they defines the values to be transferred to a method. The values that you transfer when the method is called, is referred to as the *actual parameters*. Above I have shown how to specify that a method has a variable number of parameters, which is really just a question that the compiler creates an array as the actual parameter. Methods parameters can generally be of any type, but you should be aware that primitive types and reference types are treated differently. For primitive types the transmitted values are directly copied, and that is, that the stack creates a copy of the parameters and the actual parameters are copied to these copies. This means that if a method is changing the value of a parameter that has a primitive type, then it is the value of the copy on the stack that is changed, and after the method is terminated, then the values of the calling code are unchanged. Wee therefore also call a parameter of a primitive type for a value parameter. If you for example consider the following method

```
public ArrayList<Course> getCourses(int year)
```

so is its parameter a primitive type, and if the method changes the value of *year*, it would only have effect in the method, but the change would not have effect in the code where the method was called.

Reference parameters are in principle transferred in the same way, where there on the stack is created a copy of the parameter and the current value is copied to this. However, you should be aware of what is being created and copied. In the case of a reference parameter what is created on the stack is a reference, and what is copied is the reference to the current object. As an example the following method has a reference parameter:

```
public void add(Course course) throws Exception
```

If the method creates a new *Course* object and sets the parameter *course* to refer to this object, it is still the copy on the stack that change, and the calling code will still refer to the old object. If the method not creates a new *Course* object it could use the *course* object's methods to change the object's state, as an example it could assign the object a new score. If you do the object that is changed is the object referenced on the stack, that is the same as the object that the calling code refers.

The effect of, that a method changes the value of a parameter, is thus different depending on whether it is a reference parameter or value parameter. A good example is a swap method, and thus a method that must reverses the two values. Consider the following method

```
public static void swap(int a, int b)
{
 int t = a;
 a = b;
 b = t;
}
```

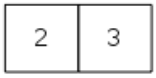
which has two parameters that are value types. If the method is executed as follows:

```
int[] arr = { 2, 3};
swap(arr[0], arr[1]);
System.out.println(arr[0] + " " + arr[1]);
```

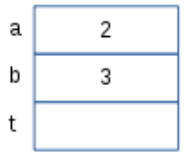
the last statement prints

```
2 3
```

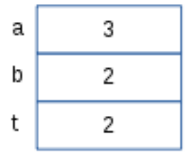
and the two numbers are not reversed. The first statement creates an array:



When the next statement calls the method *swap()*, and the variables *arr[0]* and *arr[1]* are transferred as the actual parameters. The method *swap()* has two parameters and a local variable, and has therefore three fields that are allocated on the stack, and the values of the actual parameters are copied to this:



The *swap()* method works in that it copies the value of the parameter *a* to the local variable *t*, and then copy the value of *b* to *a*. Finally the value of *t* is copied to *b*, and then the contents of the stack is as shown below. This means that the two values are reversed, but it happened on the stack, and after the method is terminated the three elements are removed from the stack, and the changes are lost. This means that the array in the calling code is unchanged.



The question is how, in Java to write a *swap()* method to swap two primitive values, and it can not actually be done directly. It is necessary to embed the values in a an object of another reference type. One solution would be:

```
public static void swap(int[] a)
{
 int t = a[0];
 a[0] = a[1];
 a[1] = t;
}
```

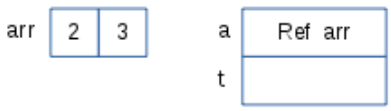
Here, the parameter is an array, which is a reference type. The algorithm is the same, and it is clear the method reverses the values of *a[0]* and *a[1]*, but it is not values on the stack. Consider the following statements where *arr* is as above:

```
swap(arr);
System.out.println(arr[0] + " " + arr[1]);
```

If they are performed you get the results

3 2

and the numbers are therefore reversed. When the method *swap()* is called, there is this time only one value to copy to the stack, which is the reference to the array *arr*:



There is still created a local variable, but when the method is performed, it is the array reference on the stack the *swap()* method works on:

The result of all this is that in practice it may be important to have in mind, where a parameter is a primitive type or a reference type, because reference types can result in undesirable side effects.

Another important thing about reference types is that what is placed on the stack is always a reference. Above I have shown a method that prints a course:

```
private static void print(Course cource) throws Exception
{
}
```

If the method is executed

```
print(c1);
```

it is a reference to a *Course* object that is placed on the stack and not a *Course* object. It is important for a *Course* object fills much more than a reference. It is thus highly effective to transfer objects as parameters to methods.

Methods have a type or they are *void*. The fact that a method is *void* means that it does not have a value, and it is therefore not required to have a *return* statement. As an example you have the method *add()* in the class *Student*. A *void* method may well have an empty *return* statement, which then has the effect that the method is terminated. If a method has a type, it must have a *return* statement that returns a value of the same type as the method's type. It is important to note that the method can only return one value, but the type of this value in turn can be anything, including a class type or an array. As an example, the method returns *getCourses()* in the class *Student* return an *ArrayList<Course>*, and precise, it is a reference to such an object.

As can be seen from the above, a method's parameters are created when the method is called and initialized by the calling code. The parameters are removed again when the method terminates, and they live only while the method executes and they can only be used or referenced from the method itself. Wee say that the parameters scope are the method's statements.

The same applies to the variables as a method might create. They are called *local variables*. They are created when the method is called, and removed again when it terminates. Their scope is also limited to the method's statements. A local variable can be created anywhere in the method, but they are all created, however when the method is called, but if a variable is referenced by a statement before it is defined, the compiler fails. In conclusion, a method can refer to

1. instance variables in the methods class
2. parameters
3. local variables

wherein, the last two have their scope limited to the method itself. By contrast, the scope of an instance variable is limited to the object, so that the variable live as long as the object does.

2.3 OBJECTS

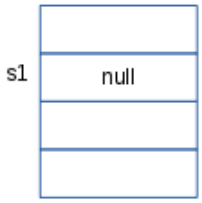
Seen from the programmer an application consist of a family of classes, but from the running program it consists of a family of objects that are created on basis of the program's classes. A class is a definition of an object in the form of instance variables that define which data the object must consist of, as well as methods defines what one can be done with an object. If you have a class such as *Subject*, you can define a variable whose type is *Subject*:

```
Subject s1;
```

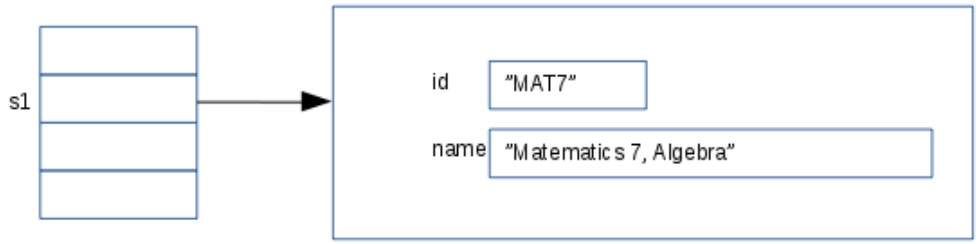
s1 is a variable as all the other variables, but it has not (yet) a value. The variable can contain a reference that you can think of as a pointer that can point to (refer to) an object. If the variable not refer to an object, its value is *null*, which merely indicates that it has no value. Objects are created with *new* as follows:

```
s1 = new Subject("MAT7", "Matematics 7, Algebra");
```

That is *new* followed by the class name and a parameter list that matches the parameters of a constructor. In this case, the class has a constructor that takes two *String* parameters, and an object can be created, as shown above. Sometimes wee say that the statment *instantiates* an object. This means that when the variable *s1* is defined, there is a variable created on the stack:



and after the object is created, the picture is the following:



When an object is created, the class's constructor is performed, and if there are several, it is the constructor whose parameters match the parameters transferred with *new*. This means that the space allocated to the object's instance variables typical are initialized with values in the constructor. In fact, the above image is not quite correct, for a *String* is an object, and the two instance variables should therefore be designed as pointers to *String* objects. I have not done that partly because strings in an application in many ways are used as if they were primitive values, and partly because the drawing better matches the way you think of a *Subject* object.

The object is created on the heap, which is a memory area in which the runtime system can continuously create new objects. When creating an object, the heap manager allocates space for object's variables, then the constructor is executed. The object then lives as long there is a reference to it, but when it is no longer the case, either because the variable that references the object is removed, or manually is set to *null*, so the object is removed by the garbage collector, and the memory that the object applied, is released and can be used for new objects. The garbage collector is a program that runs in the background and at intervals remove the objects that no longer have references.

2.4 VISIBILITY

Visibility tells where a class or its members may be used. As for classes, it's simple, when a class is defined either *public* or else you specify no visibility. A *public* class can be used anywhere, and any other class can refer to a *public* class. However, if you do not specify any visibility, the class can only be used by classes in the same *package*.

Regarding the classes members that can be variables, constructors and methods, there are four possibilities:

- *private*
- *public*
- *protected*
- no visibility

The meaning of the first two has already been explained and *protected* is a between thing where a *protected* member can be referenced from other classes in the same package and from derived classes. I have previously briefly explained inheritance, including derived classes, but the topic is described in more detail in a later chapter in this book, where I will also demonstrate the use of *protected*. Finally, there is the possibility of not indicating any visibility which means that a member can be referenced only within the same package.

It is the programmer of a class that defines visibilty for the class's members. In principle, one could make everything *public*, but it would also increase the risk that a variable or method was used in an unintended way. Therefore, you should open as little up for the class's members as possible. As mentioned earlier, do you usually defines all variables *private*, and so equips a class with the necessary *public* methods, and for the sake of derived classes you can also be considered to define methods and exceptionally variables as *protected*.

2.5 STATICAL MEMBERS

Both variables and methods can be *static*. A static variable is a variable that is shared between all objects of a class. When creating an object of a class on the heap, there is not allocated place for static variables, but they are created somewhere in memory where everyone has access to them, and if they have public visibility, it is not only objects of the class, which have access to them.

As an example of using a static variable, the class *Student* has an *id*, which is a unique number that identifies a student. When you create objects, it is often necessary that these objects can be identified by a unique key, and to ensure uniqueness, this number is automatically incremented by 1 each time a student is created. It is possible because the class has a *static* variable that contains the number of the last student created. You must note that it is necessary that this variable is *static*, since it would otherwise be created each time, you creates a *Student* object. It is, in this way of identifying objects using an auto-generated number, a technique which is sometimes used in databases. In this context, the solution is a little searched when the number is not saved anywhere and students will then possibly get new numbers the next time the program is excuted, and the purpose is indeed only to show an example of a *static* variable.

There are many other examples of static variables, and as an example I can mention the class *Cube* from the book Java 1, which had a random number generator, which was also defined as a *static* variable. There were the reason that all *Cube* objects should use the same random number generator when it is initialized by reading the hardware clock. If each *Cube* object had its own random number generator, these would possibly be initialized with the same value, thereby generating the same sequence of random numbers.

Classes can also have static methods, and in fact I have already used many examples. As an example the method *studentOk()* in the class *Student*. When a class has a static method, it can be used without having an object as the method can be referred to with the name of the class:

```
if (Student.student("poul.klausen@mail.dk", "Poul Klausen")
{
}
```

In general is a static method written as other methods, and can have both parameters and have a return value, and the same rules apply regarding visibility, but a static method can not reference instance variables – it is not associated with a specific object. You must specifically note that within the class where the method is defined, it can be referenced in the same way as any other of the class's methods.

A class may also have a *static initialize block* that can be used to initialize static variables, for example has the class Student the following block:

```
static
{
 nummer = 1000;
}
```

because by one reason or another I wants that the first student must have the key 1001. In this case there is no justification for the block, then you as well could initialized the variable directly, but in other contexts it may be important to initialize static variables otherwise than by simple assignment, for example by reading the data in a file. You should also note that the syntax is simple, and that a class can have all the static initialize blocks, as you wish. If there are more the runtime system guarantees, that they are performed in the order in which they appear in the code.

If you considers a main class, NetBeans creates the following class:

```
package students;

public class Students
{
 public static void main(String[] args)
 {
 }
}
```

which alone has a method with the name *main()*. When the program is executed, the runtime system search for a method with this name and where appropriate executes the method. As the runtime system does not create an object of the class *Students*, the method must be *static*. If the *main()* method wants to execute a method in the same class, it must generally be *static*, and the same applies if you in a static method in the main class refers to the variables in the class:

```
package students;

import java.util.*;

public class Students
{
 private static Random rand = new Random();

 public static void main(String[] args)
 {
  test00();
 }

 private static void test00()
 {
  System.out.println(rand.nextDouble());
 }
}
```

That's why I until this place has always defined members in the main class as static. If you do not want that – and it is not necessary – you must write something like the following:

```
package students;

import java.util.*;

public class Students
{
 private Random rand = new Random();

 public static void main(String[] args)
 {
  Students obj = new Students();
  obj.test00();
 }

 private void test00()
 {
  System.out.println(rand.nextDouble());
 }
}
```

This means that in *main()* method creates an object of the class itself, and using this object it can then refer to non-static methods that you can use ordinary instance variables. There are rarely any good reason for this step, and typically the main-class will consists only of static variables and static methods.

2.6 THE CURRENCYPROGRAM

I will conclude this chapter on classes with a program that can convert an amount in one currency to an amount in another currency. The program consists of several classes, but basically there is nothing new regarding classes, but the program introduces the concept of design patterns and in respect of two simple patterns. A design pattern is a certain way to solve certain problems that are general in nature and appearing in many different situations. It is natural to seek a standard for how to use proven methods for solving such a problem, and that's what a design pattern is. In the example I presents the patterns

1. a *singleton*
2. an *iterator pattern*

The program has a simple model class, which is called *Currency* that represents a currency with three properties:

- currency code (that is a code on 3 characters)
- currency name (that must not be blank)
- currency rate that should be a non-negative number

The code of the class is shown below incl. the most important comments and do not require further explanation:

```
package currencyprogram;

/**
 * Class that represents a currency when the currency rate is relative to
 * Danish crowns.
 */
public class Currency
{
 private String code;  // currency code
 private String name;  // currency name
 private double rate;  // currency rate
```

```
public Currency(String code, String name) throws Exception
{
 this(code, name, 0);
}

public Currency(String code, String name, double rate) throws Exception
{
 if (!valutaOk(code, name, rate)) throw new Exception("Illegal currency");
 this.code = code;
 this.name = name;
 this.rate = rate;
}

public String getCode()
{
 return code;
}

public String getName()
{
 return name;
}
```

```
 public void setName(String name) throws Exception
 {
  if (name == null || name.length() == 0)
   throw new Exception("Illegal currency name");
  this.name = name;
 }

 public double getRate()
 {
  return rate;
 }

 public void setRate(double rate) throws Exception
 {
  if (rate < 0) throw new Exception("Illegal currency rate");
  this.rate = rate;
 }

 public String toString()
 {
  return code + ": " + name;
 }

 /**
  * Validates the values of a currency where the code must be three characters,
  * the name must not be blank and the currency rate must non-negative.
  * @param code Currency code
  * @param name Currency name
  * @param rate Currency rate
  * @return true, If the three values results in a legal currency, else false.
  */
 public static boolean valutaOk(String code, String name, double rate)
 {
  return code != null && code.length() == 3 && name != null && name.length() > 0
    && rate >= 0;
 }
}
```

You should note that the above class is a typical model class that describes an object in the program's problem area.

THE SINGLETON PATTERN

The next class is called *CurrencyTable*, and is a class that defines the *Currency* objects that the program knows and has to work with. The program must have an object of the type *CurrencyTable*, and it is an object, which in principle should always be there and be available no matter where in the program you are. Since it is very often that a situation arises that a program must use an object of a particular type, and you want to ensure

1. the object is always there, without explicitly being created
2. the object is available to all other objects in the program
3. there certainly exists only one object of that type

there has been defined a particular design pattern for how the class to such an object should be written. This design pattern is called a *singleton*. The class can be written as follows:

```
package currencyprogram;

import java.util.*;

/**
 * Class which represents a currency table. The class is implemented as a
 * singleton.
 * Data for the currencies is laid out in a table at the end of the code.
 */
public class CurrencyTable implements Iterable<Currency>
{
 private static CurrencyTable instance = null;     // an instance of the class

 private ArrayList<Currency> table = new ArrayList(); // ArrayList to currencies

 // Private constructor to ensure that the class can not be instantiated
 // from other classes.
 private CurrencyTable()
 {
  init();
 }

 public static CurrencyTable getInstance()
 {
  if (instance == null)
  {
   synchronized (CurrencyTable.class)
   {
    if (instance == null) instance = new CurrencyTable();
```

```
   }
  }
  return instance;
}

public Currency getCurrency(String code) throws Exception
{
  for (Currency c : table) if (c.getCode().equals(code)) return c;
  throw new Exception("Illegal currency");
}

public Iterator<Currency> iterator()
{
  return table.iterator();
}

/**
  * Updating the currency table with a currency. If this currency is already
  * in the table the name and rate are updated. Otherwise, add a new currency
  * to the table.
  * @param currency The currency
  * @return true, if the table was updated correctly
  */
```

```
public boolean update(Currency currency)
{
 for (Currency c : this)
  if (c.getCode().equals(currency.getCode()))
  {
   try
   {
    c.setName(currency.getName());
    c.setRate(currency.getRate());
    return true;
   }
   catch (Exception ex)
   {
    return false;
   }
  }
 table.add(currency);
 return true;
}

private void init()
{
 for (String line : rates)
 {
  StringTokenizer tk = new StringTokenizer(line, ";");
  if (tk.countTokens() == 3)
  {
   try
   {
    String name = tk.nextToken();
    String code = tk.nextToken();
    double rate = Double.parseDouble(tk.nextToken());
    table.add(new Currency(code, name, rate));
   }
   catch (Exception ex)
   {
    System.out.println(ex.getMessage() + "\n" + line);
   }
  }
  else System.out.println("Error: " + line);
 }
}

private static String[] rates =
{
 "Danske kroner;DKK;100.00",
 "Euro;EUR;746.00",
```

```
  "Amerikanske dollar;USD;674.44",
  "Britiske pund;GBP;1034.10",
  "Svenske kroner;SEK;79.31",
  "Norske kroner;NOK;79.68",
  "Schweiziske franc;CHF;685.66",
  "Japanske yen;JPY;5.6065",
  "Australske dollar;AUD;487.61",
  "Brasilianske real;BRL;171.98",
  "Bulgarske lev;BGN;381.43",
  "Canadiske dollar;CAD;510.19",
  "Filippinske peso;PHP;14.40",
  "Hongkong dollar;HKD;87.02",
  "Indiske rupee;INR;10.38",
  "Indonesiske rupiah;IDR;0.0494",
  "Israelske shekel;ILS;174.48",
  "Kinesiske Yuan renminbi;CNY;106.17",
  "Kroatiske kuna;HRK;97.87",
  "Malaysiske ringgit;MYR;157.78",
  "Mexicanske peso;MXN;40.65",
  "New Zealandske dollar;NZD;458.77",
  "Polske zloty;PLN;173.82",
  "Rumænske lei;RON;168.13",
  "Russiske rubel;RUB;10.42",
  "Singapore dollar;SGD;483.72",
  "Sydafrikanske rand;ZAR;49.08",
  "Sydkoreanske won;KRW;0.5933",
  "Thailandske baht;THB;19.00",
  "Tjekkiske koruna;CZK;27.53",
  "Tyrkiske lira;TRY;231.78",
  "Ungarske forint;HUF;2.385"
 };
}
```

This time there are more details to note. The class has an instance variable, which is called *table* and is an *ArrayList<Currency>* and should be used for the *Currency* objects. The class also has a static variable named *instance* that is initialized to *null*. This variable is an important part of the singleton pattern. The class has a constructor, but you should note that the constructor is defined *private*, and when the class has no other constructors, it means that other objects can not instantiate objects of this class. It is another important part of the singleton pattern.

The *ArrayList* must be initialized with *Currency* objects, and it requires currency data in the form of a rate list. It can, for example be found on

http://www.nationalbanken.dk/da/statistik/valutakurs/Sider/Default.aspx

and an example is laid out in the bottom of the class as an array of strings. The method *init()* uses this the array to create *Currency* objects, and the *init()* method is called from the *private* constructor. It is of course a little pseudo, since it is not current exchange rates, but if you wants you can update the table.

The class has a static method called *getInstance()*, which returns the static variable *instance*. It is the last and perhaps most important part of the singleton pattern. The method works in that way that it tests whether the variable *instance* is *null*. If so, it creates a *CurrencyTable* object and assigns the result to *instance*. It is possible, for the method *getInstance()* is a member of the class *CurrencyTable* and can therefore refer to the private constructor. Finally the method returns the variable *instance*, and thus a reference to a *CurrencyTable* object. Other objects can refer to the *CurrencyTable* object by this reference, and when the object can not be created otherwise, there is exactly one object referenced. You should note that the line that instantiates the object is placed in a *synchronized* block. Preliminary simply accept it, but it is important for programs that creates multiple threads.

THE ITERATOR PATTERN

In terms of the other methods in the class, there is nothing to explain except the method *iterator()*. The class is an example of a collection and, in this case a collection of *Currency* objects. That kind of collections with objects of a certain kind occur very often in practice. One of the operations that almost always is needed is being able to iterate over the collection with a loop. This requires access to the objects and it is often resolved by means of a pattern which we call the *iterator pattern*. The class *CurrencyTable* implements this pattern as follows:

The class implements an interface, which here is called *Iterable<Currency>*. This means that the class must implements a method with the following signature:

```
public Iterator<Currency> iterator()
```

where *Iterator<Currency>* is an interface that defines methods, so you can iterate over the collection of *Currency* objects. In this case it is particularly simple, since an *ArrayList* has such an iterator, and the method can therefore be written as follows:

```
public Iterator<Currency> iterator()
{
 return table.iterator();
}
```

How to even write an iterator I will return to later, but the result of the iterator pattern is that you can use the extended *for* construction:

```
for (Currency c : CurrencyTable.getInstance()) { … }
```

In reality it is nothing but a short way of writing

```
for (Iterator<Currency> itr = CurrencyTable.getInstance().iterator();
  itr.hasNext(); )
{
 Currency c = itr.next();
 …
}
```

The two classes *Currency* and *CurrencyTable* are the program's model.

THE CONTROLLER

The program has a controller class that by using the above model classes must perform the currency conversion, including validation of parameters from the user interface:

```
package currencyprogram;
import java.util.*;
import java.io.*;
public class Controller
{
 public double calculate(String amount, Currency from, Currency to)
  throws Exception
 {
  try
  {
   return calculate(Double.parseDouble(amount), from, to);
  }
  catch (Exception ex)
  {
   throw new Exception("Illegal amount");
  }
 }

 public double calculate(double amount, Currency from, Currency to)
  throws Exception
 {
  if (from == null || to == null) throw new Exception("No currency");
  return amount * from.getRate() / to.getRate();
 }

 public ArrayList<String> update(File file)
 {
  ArrayList<String> errors = new ArrayList();
  try
  {
   BufferedReader reader = new BufferedReader(new FileReader(file));
   for (String line = reader.readLine(); line != null; line = reader.readLine())
   {
    StringTokenizer tk = new StringTokenizer(line, ";");
    if (tk.countTokens() == 3)
    {
     try
     {
      String name = tk.nextToken();
      String code = tk.nextToken();
      double rate = Double.parseDouble(tk.nextToken());
      CurrencyTable.getInstance().update(new Currency(code, name, rate));
     }
```

```
      catch (Exception ex)
      {
       errors.add(line);
      }
     }
     else errors.add(line);
    }
    reader.close();
   }
   catch (Exception ex)
   {
    errors.add(ex.getMessage());
   }
   return errors;
  }
}
```

There are two overloadings of the method *calculate()*, which is the method that will perform the conversion. The difference is only the type of the first parameter, wherein the first converts this from *String* to a *double*. Furthermore, there is a method *update()*, which parameter is a *File* object. The file will represent a text file that contains a list of currencies when the file must have the same format as the table been laid out in the class *CurrencyTable*. The method updates the model corresponding to the content of the file. The method returns an *ArrayList* which contains the lines from the file that could not be successfully parsed into a *Currency*.

MAINCONSOLE

Regarding the program itself it is really two programs. The first is a console application that works as follows:

The program starts to print an overview of the current currencies. In turn, there is a loop in which the user for each iteration must

1. Enter the currency code for the currency to be converted from
2. Enter the currency code for the currency to be converted to
3. Enter the amount to be converted

and then the program prints the result of the conversion. This dialogue is repeated until the user just types *Enter* for the first currency code. The program is represented by the class *MainConsole*. I will not show the code here, but when you study the code, notice how I use that class *CurrencyTable* is a singleton, and that it implements the iterator pattern.

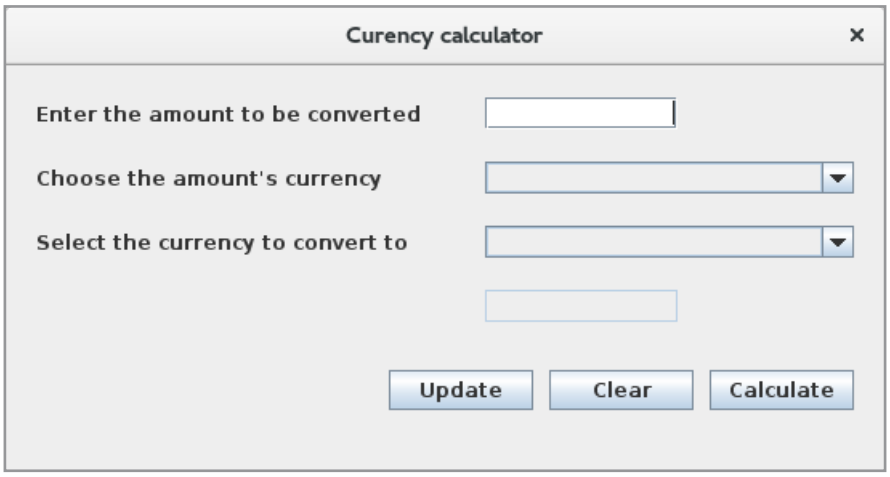
You should note that compared to other console applications that I've looked at the code is moved from the main class of its own class.

MAINVIEW

The other program is a GUI program that opens the window shown on the next page. To make a currency calculation the user must enter the amount and select two currencies and then click the *Calculate* button. The view then calls the controller to do the calculation and use the result to update a field with the calculated value. If the user clicks the *Clear* button, all fields will be cleared, and the combo box's has no selection.

If the user clicks the *Update* button the program opens a file dialog so the user can browse the file system for a semikolon delimited file with currencies rates. The file is sent to the controller that use the file to update the model's *CurrrenceTable*.

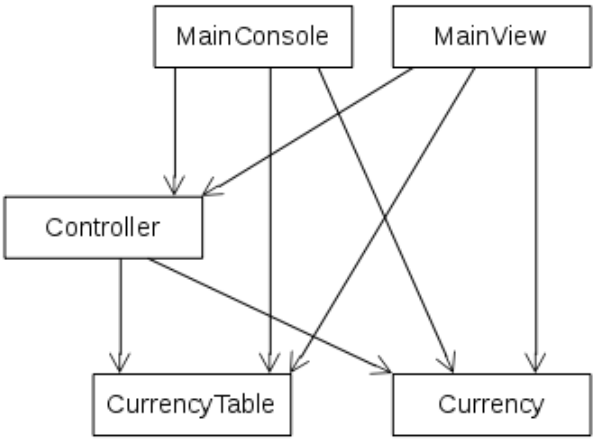
Again, I will not show the code, since it contains nothing new, but the window class is called *MainView*.



THE DESIGN

The goal of the two versions of the program is to demonstrate why it is important to separate an application in well-defined modules or layers. In this case, the program's data arre defined by two model classes that is *Currency* and *CurrencyTable*, while the data processing is carried out in the class *Controller*. The difference between the two programs is alone the view layer that in the one case is a console window, while in the second case it is a GUI window.

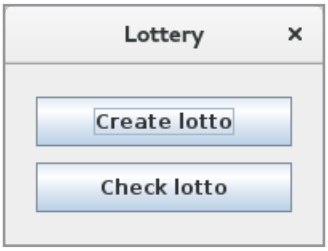
The relationship between the program's classes can be illustrated as follows, where the arrows shows which classes know who



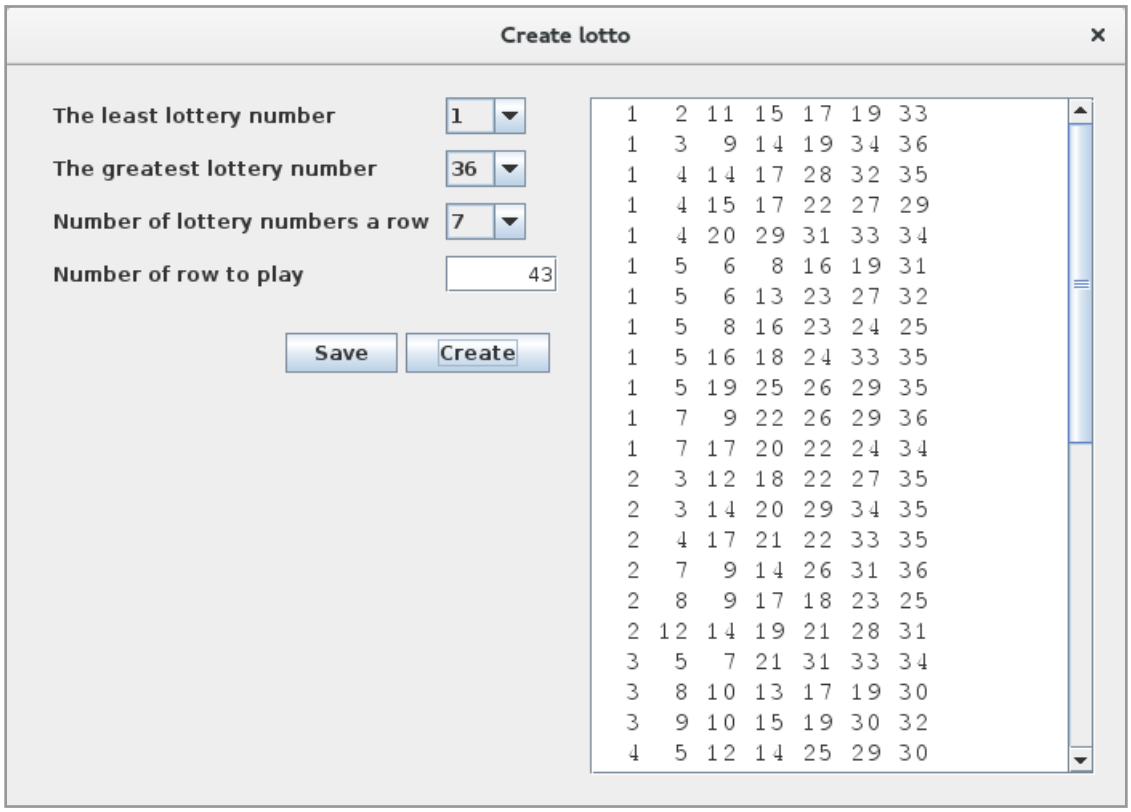
That is the model layer's classes do not know neither the controller layer or view layer and the controller layer knows nothing about the view layer. The result is that you can replace the view layer without it affects the rest of the program.

PROBLEM 2

In the book Java 1 I write a program that could create lottery rows and validates these rows again when the week's lottery is ready. The program was written as a command where you defines by parameters as arguments on the command line that tells what the command shoul do. You must now write a similar program, but such that the program this time has a graphical user interface. When you opens the program, you should get a simple window where you can choose between the program's two functions:

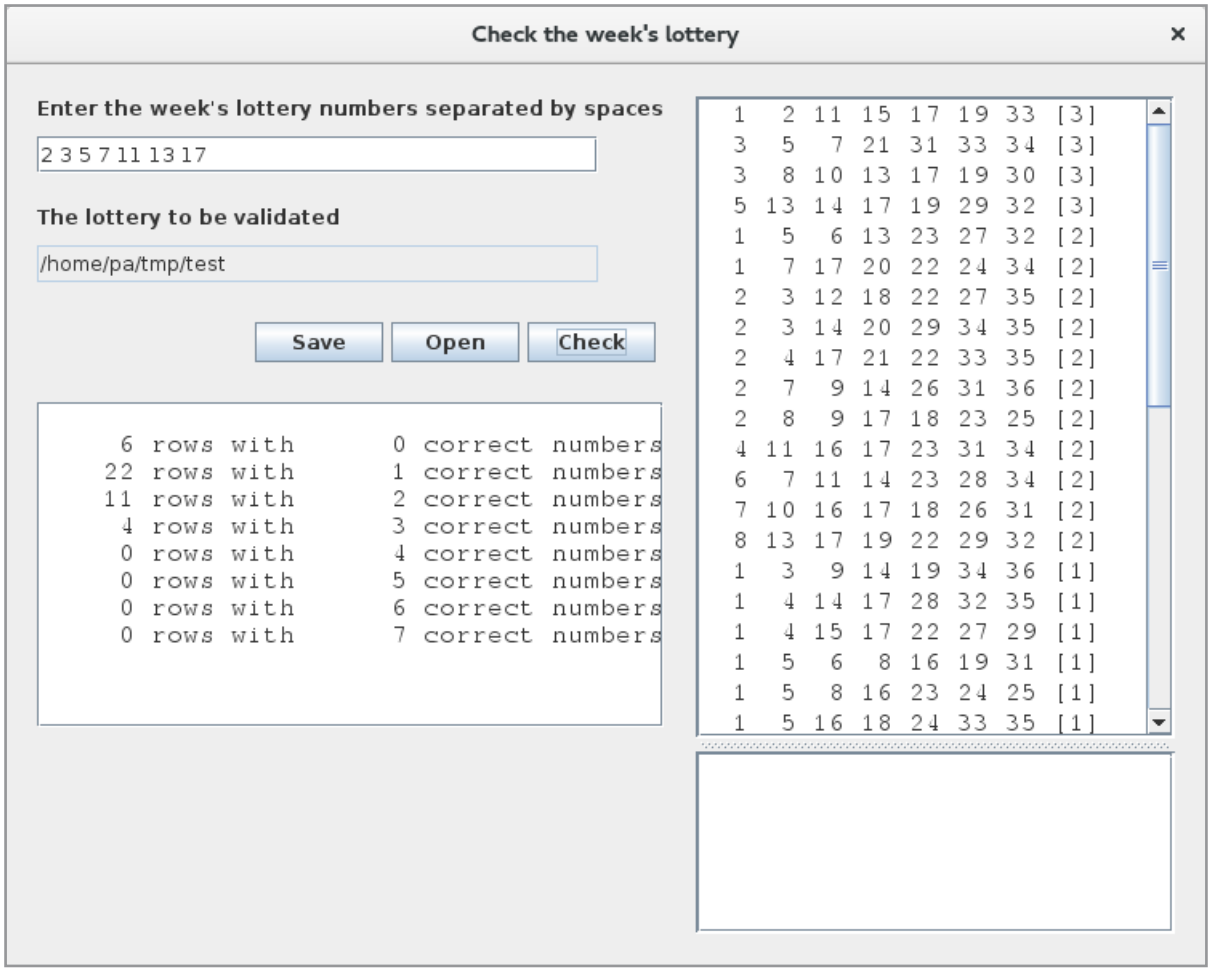


If one here click on the first button, the program should open the following window, that is used to create a new lottery and thus generate new lottery rows:



The lottery rows appears in a list box whose content can then be saved to a file.

If you click on the second button, the program shoul open the window below that is used to validate a lottery against the week's lottery numbers. The window has a text box to enter the week's lottery numbes and another field to shows the result of a control, and there are two list boxes to the right, which respectively should show the validated rows and possible rows with error.



You should solve the task by the largest possible extent of reuse of class/code from the solution in Java 1. The class's *LottoNumber*, *LottoGame* and *LottoRow* should be used as the are. You should write a controller to each of the two windows, and the code for these controller class's should be found in the class *LottoCoupon*. After you have written the controller class's you can remove the class *LottoCoupon* from the project.

3 INTERFACES

This chapter describes interfaces and in the next chapter I describes inheritance that is two other object-oriented concepts already briefly mentioned in Java 1, and in fact I has in Java 2 used both concepts without explicitly draw attention to it. An interface defines the properties that a class must have, while inheritance is a question about how to extends a class with new properties in terms of new instance variables or methods. Immediately the two thing do not seems having much to do with each other, but they have largely, and therefore I treated both concepts subsequent.

To illustere both concepts I need some examples, and I want to use the same examples as in the previous chapter, namely classes concerning students, and classes concerning books in a library, and much of what follows will also address how these classes can be modified.

3.1 INTERFACES

Technically, an interface is a type, and it is a reference type. You can therefore do basically the same with an interface as with a class, except that an interface can not be instantiated. You can not create an object whose type is an interface, but otherwise an interface may be used as a type for parameters, return values and variables.

Conceptually, an interface is a contract, and an interface can tell that an object has a specific property. The interface specifies only what you can do with the object, but not how that behavior is implemented, and that's exactly what you want to achieve. Who that has to use the object, only know it through the defining interface (that is the contract), but do not know anything about how the class that underlies the object is made. So long this class comply with the contract – and does not change the interface – the class can be changed without it affects the code that uses the object.

Typically, an interface has only signatures of methods (but there are other options, as explained below). An interface is defined by almost the same syntax as a class, but with the difference that the word *class* is replaced by the word *interface*. I will, as mentioned, often use the convention that I let the name of an interface start with a big I. In the previous chapter I defined a the class *Subject*, representing a teaching subject and thus a concept that can be included in a program concerning education. The concept can be defined as follows (see the project *Students3*):

```
package students;
/**
 * Defines a subject for an education.
 */
public interface ISubject
{
 public String getId();

 /**
  * @return The subjects name
  */
 public String getName();
 /**
  * Changes the subject's name. The subject must have a name, and if the parameter
  * does not specify a name the method raises a exception.
  * @param name The subjects name
  * @throws Exception If the name is null or blank
  */
 public void setName(String name) throws Exception;
}
```

The interface is called *ISubject*, and the only thing you can read is that a concept that is a *ISubject*, has three methods where the comments explains what these methods do.

Once you have the interface, a class can implement this interface:

```
public class Subject implements ISubject
{
```

and it is the only change in the the class *Subject*, because the class has (implements) the three methods defined by the interface. So the question is what you have achieved with that and so far nothing, but I will make some changes in the *Course* class. It has a variable of the type *Subject*, and I will change the definition of this variable, so that it instead has the type *ISubject*:

```
private ISubject subject;
```

When a class implements an interface, the class's type is special the type of this interface, and therefore it makes sense to say that a *Subject* object is also an *ISubject*. I have also changed all the parameters of constructors and methods whose type is *Subject*, as their type now are *ISubject*. You should note that the program still can be translated and run. The result is that the class *Course* no longer knows the class *Subject*, but it knows only subject objects through the defining interface *ISubject*. A *Course* know what it can with a subject (it knows the contract), but not how a subject is implemented. The two classes are now more loosely coupled than they were before, and that means that you get a program that is easier to maintain, as you can change the class *Subject* without it matters for classes that use *Subject* objects.

The fact that in this way defines the classes by means of an interface, and other classes only know a class through its interface is a principle or pattern, commonly referred to as *programming to an interface*.

An interface in addition to signatures of methods can also contain static variables and static methods – they are not attached to an object. The class *Subject* has a method *subjectOk()*, which tests whether the values in a subject are legal. It is a static method and does not depend on a *Subject* object, and it can therefore be moved to the interface. This means, however, that in the class *Subject*, the reference to the method (there are two) must be changed to

```
ISubject.subjectOk(....)
```

where you in front of the method's name has to write name the type that defines the method.

In the same way a *Course* can be defined by an interface:

```
package students;

/**
 * Defines a course for a specified year for a specific subject.
 * A course is associated with a particular student, and a course represent
 * a subject that a student has completed.
 * It is an assumption that the same subject only can be taking once a year.
 */
public interface ICourse

{
 /**
  * A cource is identified by the subjects id and the year
  * @return Course ID composed of the year, the subject's id separated by a hyphen
  */
 public String getId();

 /**
  * @return The year where the course is held.
  */
 public int getYear();
```

```
  /**
   * @return true, if the student has completed the course
   */
  public boolean completed();

  /**
   * @return The character that the student has achieved
   * @throws Exception If a student has not obtained a character
   */
  public int getScore() throws Exception;

  /**
   * Assigns this course a score.
   * @param score The score that is the obtained
   * @throws Exception If the score is illegal
   */
  public void setScore(int score) throws Exception;

  /**
   * Assigns this course a character.
   * @param score The score that is the obtained
   * @throws Exception If the score is illegal
   */
  public void setScore(String score) throws Exception;

  /**
   * @return Returns the subject for this course
   */
  public ISubject getSubject();

  /**
   * Tests whether the parameters for a course are legal
   * @param year The year for the course
   * @param subject The subject that this course deals
   * @return true, If the year is illegal or the subject is null
   */
  public static boolean courseOk(int year, ISubject subject)
  {
   return year >= 2000 && year < 2100 && subject != null;
  }
}
```

There is not much to explain, but you should note two things:

1. there is defined a new method *getSubject()*, which is not part of the class *Course*
2. the method *toString()* is not defined, and it was nor defined in the interface *ISubject*

The class *Course* must implements the interface *ICourse*, and because of the first of the above observations, it is necessary to add a new method. A class that implements an interface must implement all the methods that the interface defines. Below i show the class *Course* that now implements the interface *ICourse* where I have deleted all comments:

```
package students;

public class Course implements ICourse
{
 private int year;
 private ISubject subject;
 private int score = Integer.MIN_VALUE;

 public Course(int year, ISubject subject) throws Exception
 {
  if (!ICourse.courseOk(year, subject)) throw new Exception("Illegal course");
  this.year = year;
  this.subject = subject;
 }

 public Course(int year, String id, String name) throws Exception
 {
  subject = new Subject(id, name);
  if (!ICourse.courseOk(year, subject)) throw
new Exception("Illegal year");
  this.year = year;
 }

 public String getId()
 {
  return year + "-" + subject.getId();
 }

 public int getYear()
 {
  return year;
 }

 public ISubject getSubject()
 {
  return subject;
 }
```

```
public boolean completed()
{
 return score > Integer.MIN_VALUE;
}

public int getScore() throws Exception
{
 if (score == Integer.MIN_VALUE)
  throw new Exception("The student has not completed the course");
 return score;
}

public void setScore(int score) throws Exception
{
 if (!scoreOk(score)) throw new Exception("Illegal ckaracter");
 this.score = score;
}

public void setScore(String score) throws Exception
{
 try
 {
  int number = Integer.parseInt(score);
```

```
    if (!scoreOk(number)) throw new Exception("Illegal score");
    this.score = number;
   }
   catch (Exception ex)
   {
    throw new Exception("Illegal score");
   }
  }

  public String toString()
  {
   return subject.toString();
  }
  private boolean scoreOk(int score)
  {
   return true;
  }
}
```

You should note:

- *implements* and then the syntax to implements an interface
- there is added a new method *getSubject()* that returns a *ISubject*
- that the method *courseOk()* is removed and moved to the interface
- that the reference to *courseOk()* in the constructor is changed

After these changes, the program can be translated and run.

If you have to complies with the principle of programming to an interface, the class *Student* must be altered so that all instances of the type *Course* are changed to *ICourse*, but then the class *Student* is also decoupled from the class *Course* and know only a course by the defining interface *ICourse*.

An interface can in the same way as a class be public or it can be specified without visibility. In the latter case, the interface is known only within the package to which it belongs. In the above examples, everywhere I have defined methods in an interface as *public*, but they are by default, even if you do not write it. I prefer always to write the word *public*, as it clarifies the methods visibility. If an interface defines data (contains variables and see possibly the interface *IPoint*) these are always

```
public static final
```

whether you write it or not. Again, I prefer to write it all, thus clarifying that it is a *public* constantly.

I would then add a small change to the class *Student*. The class has a static method *studentOk()*, which validates the name and email address of a student to test where a student is legal. The controls are quite trivial, since the method only tests whether the value is specified for both the name and address. I will now extend the control, so the method also tests whether the mail address in the correct format:

```
public static boolean studentOk(String mail, String name)
{
 return mailOk(mail) && name != null && name.length() > 0;
}

private static boolean mailOk(String mail)
{
 if (mail == null || mail.length() == 0) return false;
 String pattern =
   "^[a-zA-Z0-9.!#$%&'*+/=?^_`{|}~-]+@((\\[[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}
   \\.[0-9]{1,3}\\])|(([a-zA-Z\\-0-9]+\\.)+[a-zA-Z]{2,}))$";
 Pattern p = Pattern.compile(pattern);
 Matcher m = p.matcher(mail);
 return m.matches();
}
```

Here, the method *mailOk()* is a method to validate whether a string may be a mail address. You must at this place just accept that method does, but it happens by using a so-called regular expressions, as discussed later in Java 4. You should note that the method is *private*, since the control of an address is not a natural property of a student. It is thus a helper method.

I've also changed the method *setMail()*:

```
public void setMail(String mail) throws Exception
{
 if (!studentOk(mail, navn)) throw new Exception("Illegal mail address");
 this.mail = mail;
}
```

Note that I here directly could have used *mailOk()*, but for the sake of the next I have not.

Also a student can be defined by an interface *IStudent*, but I will not show the entire code here since it does not contain anything new. However, there is one thing which creates a challenge. I want to move the method *studentOk()* to the interface, and since it is a static method, you can immediately do it, but the private method *maikOk()* should be moved to, and it gives a problem since an interface can not have a private method.

An interface may have an inner class, and although it is a subject which first addressed later, it's just a question that there is a class within an interface. The method *studentOk()* can thus be moved to the interface as follows:

```
package students;

import java.util.*;
import java.util.regex.*;
/**
 * The interface defines a student when a student is characterized by an
 * identifier, a name and a mail address.
 * A student also has a list of courses that the student has completed or
 * participate in.
 */
```

```
public interface IStudent
{
 …

 /**
  * Tester om mail og navn kan repræsentere en studerende.
  * @param mail En studerendes mailadresse
  * @param navn En studerendes navn
  * @return true, hvis mail og navn repræsenterer en lovlig studerende
  */
 public static boolean studentOk(String mail, String navn)
 {
  return Email.mailOk(mail) && navn != null && navn.length() > 0;
 }

 class Email
 {
  private static boolean mailOk(String mail)
  {
   String pattern = "^[a-zA-Z0-9.!#$%&'*+/=?^_`{|}~-]+@((\\[[0-9]{1,3}\\.[0-9]
    {1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}\\])|(([a-zA-Z\\-0-9]+\\.)+[a-zA-Z]{2,}))$"
   Pattern p = Pattern.compile(pattern);
   Matcher m = p.matcher(mail);
   return m.matches();
  }
 }
}
```

The important of this example is to show that an interface can have an inner class.

Classes can implement an interface, which as stated corresponds to the class comply a contract. Classes, however, can implements multiple interfaces and thus comply with several contracts. If the class implements several interfaces the syntax is to list them as a comma separated list after the word *implements*. As an example, the following interface defines points in the ECTS system:

```
package students;
/**
 * Defines how many ECTS assigned to a subject.
 */
public interface IPoint

{
 /**
  * Number of points a year
  */
 public static final int AAR = 60;
```

```
 /**
  * @return The subject's ECTS
  */
 public int getECTS();

 /**
  * Changes the subtect's ECTS
  * @param ects The subject's ECTS
  * @throws Exception The number must not be negative
  */
 public void setECTS(int ects) throws Exception;
}
```

The interface is not particularly interesting, as it only in principle defines an integer, but you might assume that a subject must have attached an ECTS, and the class *Subjects* could then be written as follows (where I again has removed all comments):

```
package students;

public class Subject implements ISubject, IPoint
{
 private String id;    // the subject id
 private String name;  // the subject's name
 private int ects = 0;  // the subtect's ECTS

 public Subject(String id, String name) throws Exception
 {
  this(id, name, 0);
 }

 public Subject(String id, String name, int ects) throws Exception
 {
  if (!ISubject.subjectOk(id, name))
   throw new Exception("The subject must have both an ID and a name");
  if (ects < 0) throw new Exception("ECTS must be non-negative");
  this.id = id;
  this.name = name;
  this.ects = ects;
 }

 public String getId()
 {
  return id;
 }
```

```
public String getName()
{
 return name;
}

public void setName(String name) throws Exception
{
 if (!ISubject.subjectOk(id, name))
  throw new Exception("The subject must have a name");
 this.name = name;
}

public int getECTS()
{
 return ects;
}

public void setECTS(int ects) throws Exception
{
 if (ects < 0) throw new Exception("ECTS must be non-negative");
 this.ects = ects;
}
```

```
 public String toString()
 {
  return name;
 }
}
```

The class now implements two interfaces, and must therefore implement the two methods, the new interface defines. Also, I have expanded the class with a new constructor. You should note the use of *this* where til old constructor calls the new constructor. You should note, that after the class *Subject* is changed the program can still be translated and executed. A *Subject* object still has the type *ISubject*, but it also has the type *IPoint*, but it is not used.

EXERCISE 4

Start by creating a copy of the project *Library3* and call the copy for *Library4*. Open the copy in NetBeans.

For each of the three classes *Publisher*, *Author* and *Book*, you must write an interface, and the respective classes must implement the interfaces. Note also that it means that the method *isbnOk()* should be moved to the interface *IBook*. After the three classes are defined using interfaces, and you must modify the classes code, so the three classes is as loosely coupled as possible, and you should change the main class for something like the following:

```
public static void main(String[] args)
{
 try
 {
  IBook b1 = create("978-1-59059-855-9",
   "Beginning Fedora From Noice to Professional", 2007, 1, 519,
   new Publisher(123, "Apress"), new Author("Shashank", "Sharma"),
   new Author("Keir", "Thomas"));
  IBook b2 = new Book("978-87-400-1676-5", "Spansk Vin");
  print(b1);
  print(b2);
  b2.setReleased(2014);
  b2.setEdition(1);
  b2.setPages(335);
  b2.setPublisher(new Publisher(200, "Politikkens Forlag"));
  b2.getAuthors().add(new Author("Thomas", "Rydberg"));
  print(b2);
 }
 catch (Exception ex)
 {
  System.out.println(ex.getMessage());
 }
}
```

```
private static void print(IAuthor a) { … }

private static void print(IBook book) { … }

private static IBook create(String isbn, String title, int released, int edition,
 int pages, IPublisher publisher, IAuthor … authors) throws Exception { … }
```

3.2 MORE STUDENTS

Above, I have introduced interfaces that defines the properties of an existing class, but in practice, you usually go the other way and defines an interface that defines a concept in the program area of concern, and then you (or let others do it) can write a class that implements that interface. Above, I have introduced definitions of students and courses in the form of *IStudent* and *ICourse* and then it might be natural to define a team of students, where the team must have a name and otherwise is simply a collection of students:

```
package students;

import java.util.*;

/**
 * Interface, that defines a team of students.
 */
public interface ITeam extends Iterable<IStudent>
{

 /**
  * @return Name of the team
  */
 public String getName();

 /**
  * Add students to the team
  * @param stud The students to be added
  */
 public void add(IStudent … stud);

 /**
  * Remove a student from the team.
  * @param id Id on the student to be removed
  * @return true, if the student was found and removed
  */
 public boolean remove(int id);
```

```
 /**
  * Returns a student with a paticular id.
  * @param id The id for the student to be returned
  * @return The student identified by id
  * @throws Exception If the student is not found
  */
 public IStudent getStudent(int id) throws Exception;

 /**
  * Returns all students, where the name contains the value of the parameter.
  * @param navn The search value
  * @return All students where the name contains the search value
  */
 public ArrayList<IStudent> findStudents(String name);

 /**
  * Returns all students, that has completed in a particular course a year.
  * @param id Id of the subject
  * @param aar Year
  * @return All students who matches the search values
  */
 public ArrayList<IStudent> findStudents(String id, int year);
}
```

There is not much to say about the interface when the comments explaining the meaning of each method, but you should note that an interface can inherit another interface, and in this case the interface *ITeam* inherits the interface *Iterable<Student>*. This means that a class that implements the interface, also has to implement the iterator pattern. Below is a class that implements the interface *ITeam*:

```
package students;

import java.util.*;

public class Team implements ITeam
{
 private String name;                  // the name
 private ArrayList<IStudent> list = new ArrayList(); // the students

 public Team(String name)
 {
  this.name = name;
 }

 public Iterator<IStudent> iterator()
 {
  return list.iterator();
 }

 public String getName()
 {
  return name;
 }

 public void add(IStudent … stud)
 {
  for (IStudent s : stud) list.add(s);
 }

 public boolean remove(int id)
 {
  for (int i = 0; i < list.size(); ++i)
   if (list.get(i).getId() == id)
   {
    list.remove(i);
    return true;
   }
  return false;
 }
```

```
 public IStudent getStudent(int id) throws Exception
 {
  for (IStudent s : list) if (s.getId() == id) return s;
  throw new Exception("Student not found");
 }

 public ArrayList<IStudent> findStudents(String name)
 {
  ArrayList<IStudent> lst = new ArrayList();
  for (IStudent s : list) if (s.getName().contains(name)) lst.add(s);
  return lst;
 }

 public ArrayList<IStudent> findStudents(String id, int year)
 {
  ArrayList<IStudent> lst = new ArrayList();
  for (IStudent s : list) for (ICourse c : s.getCourses(year))
   if (c.completed() && c.getSubject().getId().equals(id)) lst.add(s);
  return lst;
 }
}
```

There is not much to explain, but you should note that the class implements the iterator pattern, and that the class does not know the class *Student*, but only know a student through the interface *IStudent*.

As the last concept in this family of types regarding students and education I will look at an institution, but this time I will not define an interface. The class should instead be written as a singleton, and the program therefore needs to know the actual class, why a defining interface is not quite as interesting. The class is called *Institution*:

```
package students;

import java.util.*;

public class Institution implements Iterable<ITeam>
{
 private static Institution instance = null;
 public final static String NAVN = "The Linux University";

 private ArrayList<ITeam> list = new ArrayList();

 private Institution()
 {
 }
```

```
public static Institution getInstance()
{
 if (instance == null)
 {
  synchronized (Institution.class)
  {
   if (instance == null) instance = new Institution();
  }
 }
 return instance;
}

public ITeam getTeam(String name) throws Exception
{
 for (ITeam t : list) if (t.getName().equals(name)) return t;
 throw new Exception("There is no team with the name " + name);
}

public void add(ITeam … teams)
{
 for (ITeam t : teams) list.add(t);
}
```

```
 public Iterator<ITeam> iterator()
 {
  return list.iterator();
 }
}
```

The class is largely self-explanatory, but you must note that it is written as a singleton, and the pattern is implemented in exactly the same way as I have shown in the currency converter. Also note that the class implements the iterator pattern and note finally that the class knows only the other program classes through there defining interfaces.

Finally, I shows the test program. The main class already has a *print()* method, which prints an *IStudent*. I have added the following method to print an *Institution*:

```
private static void print()
{
 try
 {
  System.out.println(Institution.NAME);
  System.out.println();
  for (ITeam team : Institution.getInstance())
  {
   System.out.println(team.getName());
   for (IStudent stud : team)
   {
    System.out.println();
    print(stud);
   }
   System.out.println();
  }
 }
 catch (Exception ex)
 {
  System.out.println(ex.getMessage());
 }
}
```

You should note that both *print()* methods not dependents on the specific classes. The *main()* method is as follows:

```
public static void main(String[] args)
{
 Institution.getInstance().add(
  createTeam("Team-A 2015",
   createStudent("olga.jensen@mail.dk", "Olga Jensen",
    createCourse(2015, "PRG", "Introduction to Java", 5, 7),
```

```
     createCourse(2015, "OPS", "Operating Systems", 15, 2),
     createCourse(2015, "SYS", "System Development", 5, 2)
    ),
    createStudent("harald.andersen@mail.dk", "Harald Andersen",
     createCourse(2015, "PRG", "Introduction to Java", 5, 4),
     createCourse(2015, "NET", "Computer Networks", 10, 12)
    )
   ),
   createTeam("Team-B 2015",
    createStudent("svend.hansen@mail.dk", "Svend Hansen",
     createCourse(2015, "OPS", "Operating Systems", 15, 10),
     createCourse(2015, "RRG", "Introduction to Java", 5, 0),
     createCourse(2015, "DBS", "Database Systems", 20, 7)
    ),
    createStudent("svend.frederiksen@mail.dk", "Svend Frederiksen",
     createCourse(2015, "OPS", "Operationg Systems", 15, 2),
     createCourse(2015, "NETL", "Computer Networks", 10, 10)
    )
   )
  );
  print();
}
```

adding two teams to the institution where each team has two students. Finally the method prints the institution. The method uses some helper methods to create objects:

```
private static ITeam createTeam(String name, IStudent … studs)
{
 ITeam team = new Team(name);
 team.add(studs);
 return team;
}

private static IStudent createStudent(String mail, String name, ICourse … course)
{
 try
 {
  return new Student(mail, name, course);
 }
 catch (Exception ex)
 {
  return null;
 }
}
private static ICourse createCourse(int year, String id, String name, int ects,
 int score)
{
```

```
  try
  {
   ICourse course = new Course(year, id, name);
   course.setScore(score);
   ((IPoint)course.getSubject()).setECTS(ects);
   return course;
  }
  catch (Exception ex)
  {
   return null;
  }
}
```

If you run the program you get the following result:

```
The Linux University

Team-A 2015

[1001] Olga Jensen
Introduction to Java, Score: 7
Operating Systems, Score: 2
System Development, Score: 2
```

```
[1002] Harald Andersen
Introduction to Java, Score: 4
Computer Networks, Score: 12
Team-B 2015

[1003] Svend Hansen
Operating Systems, Score: 10
Introduction to Java, Score: 0
Database Systems, Score: 7
```

EXERCISE 5

Make a copy of the project *Library4* and call the copy *Library5*. Open the copy in NetBeans. You must write an interface defining a book list:

```
package library;

import java.util.*;

/**
 * Defines a book list
 */
public interface IBooklist extends Iterable<IBook>
{
 /**
  * Method, that adds books to the book list.
  * If the list already contains a book with
the same ISBN, the book should be
  * ignored.
  * @param books The books to be added
  */
 public void add(IBook … books);

 /**
  * Method that creates a book and adds it to the list. The book is created on the
  * basis of et variabelt antal strenge, som fortolkes på følgende måde i den
  * nævnte rækkefølge:
  *  ISBN
  *  title
  *  release year
  *  edition
  *  pages
```

```
     *  publisher name
     *  an number of pairs of authors' first and last name
     * Only the first two arguments are required.
     * Publisher's number extracted from the ISBN (see problem 1).
     * @param elem Values to the book
     * @throws Exception If the passed values are not legal book data
     */
    public void add(String … elem) throws Exception;

    /**
     * Return the book with a certain isbn.
     * @param isbn ISBN of the book to be returned
     * @return The book idenfificeret of isbn
     * @throws Exception If the book is not found
     */
    public IBook getBog(String isbn) throws Exception;

    /**
     * Search books in the book list and returns a list of the books that matches the
     * search values.
     * The list is searched combined so that a book matching the search values if it
     * matches all search values.
     * If a String is null or blank, the criterion is ignored. The same applies to an
     * int that is 0.
     * By searching on strings that should not be distinction between uppercase and
     * lowercase letters, and a criterion matches if the book value contains the
     * search string.
     * Especially by searching the author a book match if it has a single author,
     * whose first and last name contains the values searched.
     * @param isbn The book's ISBN, which matches the book if it is null, blank or
     *        the book's ISBN contains the value
     * @param title The book's title, which matches the book if it is null, blank or
     *        book title contains the value
     * @param year The book's publication, matching the book if 0 or release year is
     *        equal to the value
     * @param edition The book's edition that matches the book if 0 or the book's
     *         edition is equal to the value
     * @param publisher The name of the publisher, which matches the book if it is
     *          null, blank or publisher's name contains the value
     * @param firstname Matches if null, blank or the book has an author whose first
     *          name contains the value
     * @param lastname Matches if null, blank or the book has an author whose last
     *          name contains the value
     * @return All books that matches the search values
     */
    public ArrayList<IBook> find(String isbn, String title, int year, int edition,
      String publisher, String firstname, String lastname);
}
```

Write a class *Booklist* that implements the interface. When you have written the class, you must writes a test method. You can define the following array in the main program, which contains data for 10 books:

```
private static String[][] data = {
 { "978-1-59059-855-9", "Beginning Fedora From Novice to Professional",
  "2007", "1", "519", "Apress", "Shashank", "Sharma", "Keir", "Thomas" },
 { "978-0-13-275727-0",
  "A practical guide to Fedora and Red Hat Enterprise Linux", "2011",
  "6", "519", "Prentice Hall", "Mark G.", "Sobell" },
 { "978-1-4842-0067-4", "Beginning Fedora Desktop: Fedora 20 Edition",
  "2014", "1", "459", "Apress", "Richard", "Petersen" },
 { "978-0-470-48504-0", "Fedora 11 and Red Hat Enterprise Linux Bible",
  "2011", "1", "1076", "Wiley Publishing", "Christopher", "Negus",
  "Eric", "Foster-Johnson" },
 { "978-1-118-99987-5", "Linux Bible", "2015", "9", "859",
  "John Wiley & Sons", "Christopher", "Negus" },
 { "0-534-95054-X", "Understanding Data Coomunications & Networks",
  "1999", "2", "711", "Cole Publishing", "William A.", "Shay" },
 { "978-0-13-255317-9", "Computer Networks", "2011", "5", "952",
  "Prentice Hall", "Andrew S.", "Tanenbaum", "David J.", "Wetherall" },
 { "0-13-148521-0", "Structured Computer Organization", "2006", "5",
  "777", "Prentice Hall", "Andrew S.", "Tanenbaum" },
```

```
  { "978-0-321-54622-7", "Data Structures & Problem Solving Using Java",
    "2010", "4", "1011", "Addison-Wesley", "Mark Allen", "Weiss" },
  { "978-0-321-63700-0", "LINQ To Objects Using C# 4.0", "2010", "1",
    "312", "Addison-Wesley", "Troy", "Magennis" },
};
```

Next, write a test method on the basis of these data that creates a book list with 10 books and print the books whose title contains the word *Fedora*.

3.3 FACTORIES

The program *Students* now consists of classes where the coupling between the classes are defined by interfaces. None of the classes know about the other classes existence, including how these classes are implemented, but they know what you can do with objects of the specific classes, since they know the contracts. Software consists of modules (which here can be translated into classes), and it is a goal to write software that consists of as loosely coupled modules as possible. To program to an interface is an important step in that direction, but somewhere, the specific objects must be created, and in the above test program is done in the particular help methods. However, it is also the only place where to references the concrete classes.

You can move the code that creates the objects to a special class, which is called a *factory* class (a class which produces objects). I've added the following class to the program:

```
package students;

public abstract class Factory
{
 public static ISubject createSubject(String id, String name) throws Exceptior
 {
  return new Subject(id, name);
 }

 public static ISubject createSubject(String id, String name, int ects)
  throws Exception
 {
  return new Subject(id, name);
 }

 public static ICourse createCourse(int year, ISubject subject) throws Exceptio
 {
  return new Course(year, subject);
 }
```

```
 public static ICourse createCourse(int year, String id, String name)
  throws Exception
 {
  return new Course(year, createSubject(id, name));
 }

 public static IStudent createStudent(String mail, String name,
  ICourse … course) throws Exception
 {
  return new Student(mail, name, course);
 }

 public static ITeam createTeam(String name, IStudent … studs)
 {
  ITeam team = new Team(name);
  team.add(studs);
  return team;
 }
}
```

The class consists only of static methods that creates objects of a concrete type, but returns the objects as an interface types. The class is defined abstract, and this means that the class can not be instantiated.

With the class *Factory* available, the test program can be written as follows, where I have not shown the printing methods, as they are not changed:

```
public static void main(String[] args)
{
 try
 {
  Institution.getInstance().add(
   Factory.createTeam("Team-A 2015",
    Factory.createStudent("olga.jensen@mail.dk", "Olga Jensen",
     createCourse(2015, "PRG", "Introduction to Java", 5, 7),
     createCourse(2015, "OPS", "Operating Systems", 15, 2),
     createCourse(2015, "SYS", "System Development", 5, 2)
    ),
    Factory.createStudent("harald.andersen@mail.dk", "Harald Andersen",
     createCourse(2015, "PRG", "Introduction to Java", 5, 4),
     createCourse(2015, "NET", "Computer Networks", 10, 12)
    )
   ),
   Factory.createTeam("Team-B 2015",
    Factory.createStudent("svend.hansen@mail.dk", "Svend Hansen",
```

```
      createCourse(2015, "OPS", "Operating Systems", 15, 10),
      createCourse(2015, "RRG", "Introduction to Java", 5, 0),
      createCourse(2015, "DBS", "Database Systems", 20, 7)
     ),
     Factory.createStudent("svend.frederiksen@mail.dk", "Svend Frederiksen",
      createCourse(2015, "OPS", "Operationg Systems", 15, 2),
      createCourse(2015, "NETL", "Computer Networks", 10, 10)
     )
    )
   );
   print();
  }
  catch (Exception ex)
  {
   System.out.println(ex);
  }
}

private static ICourse createCourse(int year,
String id, String name, int ects,
 int score) throws Exception
{
```

```
 ICourse course =
  Factory.createCourse(year, Factory.createSubject(id, name, ects));
 course.setScore(score);
 return course;
}
```

Then there are no coupling between the program and the concrete classes. In this case, the class *Factory* is trivial with simple methods that do nothing but to encapsulate a *new* operation, but other factory classes may be more complex, for example, to read data from a file or database.

EXERCISE 6

Create a copy of the project *Library5* and call the copy *Library6*. Open the copy in NetBeans and create the following factory class:

```
public class Factory
{
 public static IPublisher createPublisher(int nummer, String name)
  throws Exception
 {
  …
 }

 public static IAuthor createAuthor(String firstname, String lastname)
  throws Exception
 {
  …
 }

 public static IBook createBook(String … elem) throws Exception
 {
  …
 }

 public static IBooklist createBooklist()
 {
  …
 }
}
```

Here are three of the methods trivial, but the method that creates a book is relatively complex. You can get the most of the required code from the class *Booklist*.

After writing the class, change the main program, so all objects in the class *Library* are created by calls to the *Factory* class. You must also change the class *Booklist*, so that it also applies the *Factory* class.

4 INHERITANCE

If you have a class and you want another class, similar to the first, but expanded with new variables or methods, or you may want one or more methods to work in a different way, the new class can inherit the first. As explained in Java 1, the class that inherits, is called for a derived class, while the class it is inherited from is called the base class. Other words for the same are respectively subclasses and superclass. I will in this section illustrate inheritance through classes, which represent loans in a bank. To make it simple, I will assume that a loan is characterized by the formation expenses and an amount that I together will call the loan's principal, an interest rate which I would assume is constant throughout the loan period, and a number of periods, which is the number of payments to repay the loan. It thus corresponds to the same requirements as I assumed in the loan calculation program in the book Java 2. Under these conditions a loan is defined as the following class:

```
package loanprogram;
public class Loan
{
 private double principal;     // the amount borrowed inc. costs
 private double interestRate; // interest rate as a number between 0 and 1
 private int periods;          // the number of periods or number of payments

 public Loan(double principal, double interestRate, int periods)
 {
  this.principal = principal;
  this.interestRate = interestRate;
  this.periods = periods;
 }

 public double getPrincipal()
 {
  return principal;
 }

 public double getInterestRate()
 {
  return interestRate;
 }

 public int getPeriods()
 {
  return periods;
 }
```

```
 public double repayment(int n)
 {
  return 0;
 }

 public double interest(int n)
 {
  return 0;
 }

 public double payment(int n)
 {
  return 0;
 }

 public double outstanding(int n)
 {
  return 0;
 }
}
```

The first three methods should be self-explanatory, and the last four returns

1. repaid by the payment of the nth payment
2. interest by the payment of the nth payment
3. the nte payment
4. the remaining debt immediately after the payment of the nth payment

If you look at the last four methods, they are all trivial and returns all 0. A loan can be repaid in several ways, and to write the code for the last four methods, it requires that you know what it is for a kind loan in question. That know the class *Loan* not, and therefore it is not able to implement these methods in a meaningful way. It may on the other hand be possible in a derived class.

The most common loan is an annuity that is characterized by the fact that you for each payment pays the same every time. When you all the time have to pay interest on what is outstanding, it means that the relationship between the repayment and the interest are changed throughout the repaid period. If b is the size of the loan (the loan principal), r is the interest rate each periode and n is the number of periods, the payment can be determined using the following formula:

$$y = \frac{br}{1-(1+r)^{-n}}$$

The outstanding debt after you have paid the kth payment can be calculated using the formula

$$rest = b(1+r)^k - y\frac{(1+r)^k - 1}{r}$$

So it is the kind of loan that I looked at in Java 2.

I will now write a class that represents an annuity when the class will inherit the class *Loan:*

```
package loanprogram;

public class Annuity extends Loan
{
 public Annuity(double principal, double interestRate, int periods)
 {
  super(principal, interestRate, periods);
 }
```

```
 public double payment(int n)
 {
  return getPrincipal() *
   getInterestRate() / (1 - Math.pow(1 + getInterestRate(), -getPeriods()));
 }

 public double outstanding(int n)
 {
  return getPrincipal() * Math.pow(1 + getInterestRate(), n) -
   payment(0) * (Math.pow(1 + getInterestRate(), n) - 1) / getInterestRate();
 }

 public double interest(int n)
 {
  return outstanding(n - 1) * getInterestRate();
 }

 public double repayment(int n)
 {
  return payment(n) - interest(n);
 }
}
```

You should note that the class inherits *Loan*, and how to specify that with *extends*. The class consists only of a constructor and the four methods which should work in a different way. Wee say that the class overrides the four methods from the base class. That the class inherits means it beyond what it itself defines also can use *public* variables and methods from the base class. Because a derived class can only refer to what in the base class is defined as *public*, it can therefore not reference the base class's variables, but it must use to the *get*-methods.

When creating an annuity object, the instance variables must be initialized, but they are in the base class, and are initialized using the base class's constructor. It is therefore necessary in one way or another to get this constructor performed. It must be called from the constructor of the class *Annuity*, but since you can not directly call a constructor, it is necessary with a special syntax:

```
super(principal, interestRate, periods);
```

that calls the base class's constructor. The statement *super* must, be the first statement in the derived class's constructor.

The biggest challenge in writing the class *Annuity* is to implement the above formulas. Here you must notice the method *Math.pow()*, which calculates the power of an argument. With this method available, it is quite simple to write the code to the four calculation methods.

If you have an object of the type *Annuity*:

```
Annuity loan = new Annuity(10000, 0.015, 10);
```

then you can write a statement like the following:

```
System.out.println(loan.repayment(5));
```

and the method is *repayment()* in the class *Annuity* is performed. You can also write

```
System.out.println(laan.getPrincipal());
```

as *loan* is an *Annuity* that inherits *Loan* and method *getPrincipal()* is available.

Sometimes it can be conveniently directly to refer to the variables in the base class from a method in a derived class, but without making them *public*. This is possible if the variables are defined *protected*:

```
public class Loan
{
 protected double principal;   // the amount borrowed inc. costs
 protected double interestRate; // interest rate as a number between 0 and 1
 protected int periods;        // the number of periods
```

A protected variable can not be referenced outside the class's package (it has package visibility), but it can be referenced from derived classes – even if they belong to a different package. As an example the class *Annuity* now can be written as follows:

```
package loanprogram;

public class Annuity extends Loan
{
 public Annuity(double principal, double interestRate, int periods)
 {
  super(principal, interestRate, periods);
 }

 public double payment(int n)
 {
  return principal * interestRate / (1 - Math.pow(1 + interestRate, -periods));
 }

 public double outstanding(int n)
 {
  return principal * Math.pow(1 + interestRate, n) -
   payment(0) * (Math.pow(1 + interestRate, n) - 1) / interestRate;
 }

 public double interest(int n)
 {
  return outstanding(n - 1) * interestRate;
 }

 public double repayment(int n)
 {
  return payment(n) - interest(n);
 }
}
```

In this context, there are no big differences, but in other situations it may be necessary to open up the protection, so derived classes can reference variables in the base class.

Note that the methods also can be *protected*.

An amortization is a table showing a summary of the payments of a loan and for each payment shows the payment, repayment, interest and outstanding debt. The following class represents an amortization:

```
package loanprogram;

public class Amortization
{
 private Loan loan;
 public Amortization(Loan loan)
 {
  this.loan = loan;
 }

 public void print()
 {
  System.out.println(
   "=========================================================================");
  System.out.printf("Principal:       %12.2f\n", loan.getPrincipal());
  System.out.printf("Interest rate:     %12.2f %%\n",
   loan.getInterestRate() * 100);
  System.out.printf("Number of periods:  %12d\n\n", loan.getPeriods());
  System.out.println(
   "Periods      Payment    Repayment    Interest  Outstanding");
  System.out.println(
   "-------------------------------------------------------------------------");
  for (int n = 1; n <= loan.getPeriods(); ++n)
   System.out.printf("%7d%16.2f%17.2f%15.2f%15.2f\n", n, loan.payment(n),
    loan.repayment(n), loan.interest(n), loan.outstanding(n));
  System.out.println(
   "-------------------------------------------------------------------------")
 }
}
```

There is not much to explain, but note that the parameter to the constructor has the type *Loan* and the class *Amortization* thus is not aware of the specific loan types such as *Annuity*. I will return to that shortly.

With the class *Amortisation* available, you can perform the following program:

```
public static void main(String[] args)
{
 new Amortization(new Annuity(10000, 0.015, 10)).print();
}
```

and the result is as shown below:

```
=====================================================================
Principal:         10000,00
Interest rate:         1,50 %
Number of periods:        10
Periods      Payment     Repayment     Interest  Outstanding
---------------------------------------------------------------------
   1      1084,34       934,34      150,00     9065,66
   2      1084,34       948,36      135,98     8117,30
   3      1084,34       962,58      121,76     7154,72
   4      1084,34       977,02      107,32     6177,70
   5      1084,34       991,68      92,67     5186,02
   6      1084,34      1006,55      77,79     4179,47
   7      1084,34      1021,65      62,69     3157,82
```

```
   8      1084,34      1036,97      47,37     2120,85
   9      1084,34      1052,53      31,81     1068,32
   10      1084,34      1068,32      16,02        0,00
-------------------------------------------------------------------------
```

In addition to the result, note that it is the methods in the class *Annuity* that are performed and it is not obvious. The class *Amortization* only know the type of *Loan* that have trivial methods, all of which returns 0. Nevertheless, it is the methods of the class *Annuity* that are executed, and it is, even if the class *Amortization* only know the type *Loan.* When such it is the runtime system that remembers that the object *loan* actually has the type *Annuity* and uses therefore the object with the right methods. It is one of the key concepts of object-oriented programming and is called *polymorphism*.

If you considers the class *Loan*, then, as mentioned above the four calculation methods are trivial and all simply returns 0. These methods are somehow meaningless, and it makes no sense to create objects whose type is *Loan*. If you did that, and sent such an object over a *Amortization* object, you would get an amortization table, where all numbers are 0. The problem can be solved by making the four methods abstract:

```
package loanprogram;

public abstract class Loan
{
 protected double principal;  // the amount borrowed inc. costs
 protected double interestRate; // interest rate as a number between 0 and 1
 protected int periods;       // the number of periods

 public Loan(double principal, double interestRate, int periods)
 {
  this.principal = principal;
  this.interestRate = interestRate;
  this.periods = periods;
 }

 public double getPrincipal()
 {
  return principal;
 }

 public double getInterestRate()
 {
  return interestRate;
 }
```

```
 public int getPeriods()
 {
  return periods;
 }

 public abstract double repayment(int n);

 public abstract double interest(int n);

 public abstract double payment(int n);

 public abstract double outstanding(int n);

}
```

An abstract method is just a prototype or a signature for a method in the same way as you defines methods in an interface. When a class contains abstract methods, the class is abstract, and it must be declared as an abstract class:

```
public abstract class Loan
```

An abstract class can not be instantiated, and you can not with *new* create an object whose type is *Loan* but an abstract class can easily be parameter to a method. That class *Loan* has four abstract methods means that anyone who knows anything that is a *Loan* knows that the object in question, has the four methods that the abstract class specify.

If the class *Loan* is changed to an abstract class as shown above, the program can still be translated and run, and it gives the same result.

One way to think of abstract classes is that you have to write a class *Loan* and know what properties and methods a *Loan* must have, but some of the methods you are not able to implement, because you do not have sufficient knowledge. These methods can then defined abstract and postpone implementation to the specific derived classes who have the necessary knowledge.

I will write another class that will represent a mix loan. It is a loan where in the first periods there is no repayment, and where you must pay interest only. After the periods without repayment the loan is an annuity, but with a fewer periods. The class may, for example be written as follows:

```
package loanprogram;
```

```
public class MixLoan extends Loan
{
 private int free;
 private Loan loan;

 public MixLoan(double principal, double interestRate, int periods, int free)
 {
  super(principal, interestRate, periods);
  this.free = free;
  loan = new Annuity(principal, interestRate, periods – free);
 }

 public int getFree()
 {
  return free;
 }

 public double payment(int n)
 {
  return n <= free ? principal * interestRate : loan.payment(n – free);
 }
```

```
 public double interest(int n)
 {
  return n <= free ? principal * interestRate : loan.interest(n - free);
 }

 public double repayment(int n)
 {
  return n <= free ? 0 : loan.repayment(n - free);
 }

 public double outstanding(int n)
 {
  return n <= free ? principal : loan.outstanding(n - free);
 }
}
```

You should note that a derived class can add new variables (in this case two) and new methods (here the method *getFree()* that returns the number of periods without repayment). The implementation of the four abstract methods are simple and are not further explained. You can test the class in *main()*:

```
public static void main(String[] args)
{
 new Amortization(new MixLoan(10000, 0.015, 10, 3)).print();
}
```

As another example of a loan, one can consider a serial loan that is a loan where the repayment is the same for every time. This means that payment is greatest at the beginning and decreases through the loan period. Just as an *Annuity* the project has a class *Serial* that is a derived class that inherits *Loan* and thus a concrete class. I will not show the code here.

The abstract class *Loan* can be defined by an interface:

```
package loanprogram;

public interface ILoan
{
 public double getPrincipal();
 public double getInterestRate();
 public int getPeriods();
 public double repayment(int n);
 public double interest(int n);
 public double payment(int n);
 public double outstanding(int n);
}
```

An interface consists only in principle of abstract methods, but it is not necessary to write the word *abstract*, but it is allowed. The word *abstract* is default.

The class *Loan* must then implement the interface *ILoan*. This means that you can delete the four abstract methods, since they are defined in the interface, but the class must still be abstract because it does not implement all of the interface methods:

```
package loanprogram;

public abstract class Loan implements ILoan
{
 protected double principal;    // the amount borrowed inc. costs
 protected double interestRate;  // interest rate as a number between 0 and 1
 protected int periods;         // the number of periods

 public Loan(double principal, double interestRate, int periods)
 {
  this.principal = principal;
  this.interestRate = interestRate;
  this.periods = periods;
 }

 public double getPrincipal()
 {
  return principal;
 }

 public double getInterestRate()
 {
  return interestRate;
 }

 public int getPeriods()
 {
  return periods;
 }
}
```

It is then possible (and you should do that) to change the class *Amortization*, where the type of the variable *loan* is changed to *ILoan,* and the same for the parameter in the constructor.

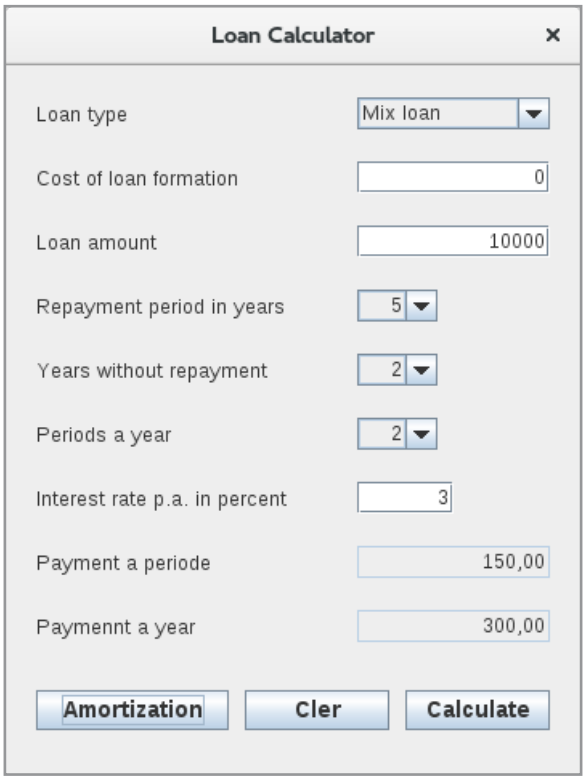
EXERCISE 7

In problem 1 in the book Java 2 you have written a loan calculation program that calculates the payment of an annuity and shows an amortization table. In this exercise you must create an expansion of this program so that it also can perform loan calculations for a mix loan and a serial loan.

Start by creating a copy of the project *LoanCalculator* from Java 2 and open the copy in NetBeans. Copy the types

- *ILoan*
- *Loan*
- *Annuity*
- *MixLaan*
- *Serial*

from the above project. Next, edit the user interface as shown below. There is added a combo box to select the type of the loan and a combo box for selecting the number of free years. If it not is a mix loan, the value of the last combo box just should be ignored. Finally, the texts of the two final fields are changed such they now respectively shows the first payment and the overall payment for the first year (the payment is not longer necessarily constant). You must then modify the model (quite a bit) so that it uses the loan types from *LoanProgram* and you must also modify the controller a bit.



PROBLEM 2

The concept of a number sequence is known from mathematics and is a sekvense of numbers. One example is the sequence of even numbers:

0, 2, 4, 6, 8, 10, …

where you constantly determines the next number by adding 2. In the same way you can consider the sequence of powers of 2 and the factorials which are

1, 2, 4, 8, 16, 32, 64, 128, …

1, 1, 2, 6, 24, 120, 720, 5040, …

At the powers of 2 you always go one step forward by multiplying by 2, while at the factorials you go a step forward by multiplying by the next positive integer. More precisely, a number sequence is a sequence of numbers

$$a_0, a_1, a_2, \ldots, a_n,$$

where there for any non negative integer is attached a (usually) real number. Taking as example the sequence consisting of the powers of 2, you sometimes write

$$\{2^n\}$$

In this sequence as example is $a_{10} = 1024$.

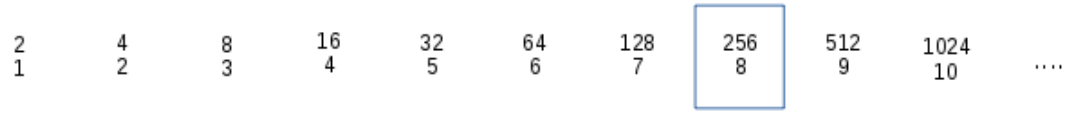
In this problem, you should write some classes that represent sequences, but only sequences whose values are integers.

Some sequences – like fatorial – is growing very rapidly, and must sequences implemented in Java, is limited in how big integers you can represent. Since you here only works with sequences with integer values, the size is limited by the type *long*. If you again consider the factorial

$$20! = 2432902008176640000$$

and it is the largest fatorial that can be represented by a *long*. Java, however, has a solution in the form of a class *BigInteger*, which is a software implementation of an integer. In principle, the class represents arbitrary large integers, but it costs obviously something since calculations on very large numbers becomes slow. Nevertheless, the class is worth knowing, so this example.

In the following you can think of a sequence, as a sequence of numbers where each number is identified by an index, but only one of these numbers are visible and is the sequence's current value. Below I have illustrated the sequence consisting of powers of 2, and there it is the number 256 with the index 8, that is visible:



Create a project called *Sequences*. Add the following interface, where the comments explains what the methods should do:

```
package sequences;

import java.math.*;

/**
 * Defines a number sequence with integer numbers where the first index is 0.
 */
public interface ISequence
{
 /**
  * @return Name of the sequence
  */
 public String getName();
 /**
  * @return The current index
  */
 public int getIndex();
```

```
 /**
  * @return The current value
  */
 public BigInteger getValue();

 /**
  * Returns the number in the sequence having the index n. At the same time,
  * this number is the actual value.
  * @param n The index the sequence must be set to
  * @return Sequence's value for the index n
  * @throws Exception If the index is less than 0
  */
 public BigInteger getValue(int n) throws Exception;

 /**
  * Steps the sequens one step forward
  */
 public void next();

 /**
  * Step the sequence one step back
  * @throws Exception If the index is 0
  */
 public void prev() throws Exception;

 /**
  * Sets the sequence of the first value that is greater than or equal to number.
  * This value is then the sequence's current value.
  * @param number The number the sequence to be set by
  */
 public void next(BigInteger number);

 /**
  * Sets the sequence of the first value that is less than or equal to number.
  * This value is then the sequence's current value.
  * @param number The number the sequence to be set by
  * @throws Exception If it not is possible to set the sequence at the value
  */
 public void prev(BigInteger number) throws Exception;
}
```

You must observe the *import* of the package *java.math*. It's the package containing the class *BigInteger*.

A sequence is characterized by eight methods, and some of these methods does not depend on a specific sequence. You should then write an abstract class *Sequence*, which implements everything that is common to all sequences. This class will then be a common base class for specific sequences.

As a next step you must implement the sequence consisting of the powers of 2. The class is a concrete sequence and should be implemented as a class that inherits *Sequence*.

Once you have written the class, you should write the following test program (in the main class):

```
public class Sequences
{
 public static void main(String[] args)
 {
  print(new Power2(), 50);
 }

 private static void print(ISequence s, int n)
 {
  try
  {
   for (int i = 0; i <= n; ++i, s.next()) System.out.println(s);
   while (s.getIndex() > 0)
   {
    s.prev();
    System.out.println(s);
   }
  }
  catch (Exception ex)
  {
   System.out.println(ex.getMessage());
  }
 }
}
```

Next, write a class that implements the fatorial, and a test class in the same way as shown above.

Then, implements and test further three sequences:

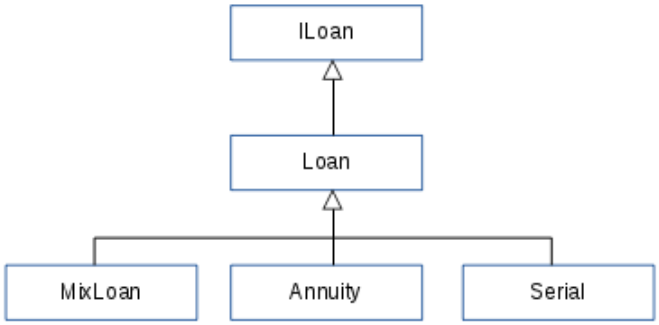
1. *Power*, where the constructor has a parameter, that is a *BigInteger*. If this parameter is called *t*, the sequence represents the numbers: $1, t, t^2, t^3, t^4, t^5, \ldots$
2. *Fibonacci*, that represents the fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …
3. *Prime*, that represents the prime numbers: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, ….

With regard to the last sequence you should examine the documentation for the class *BigInteger* as it actually has a function that with reasonable probability can test whether an integer is a prime.

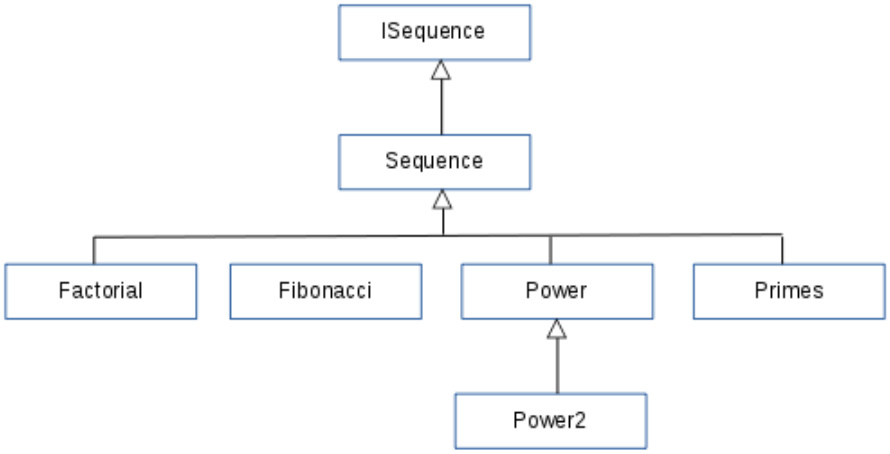
After you have implemented these three sequences and tested that they work properly, change the class *Power2* such it inherits the class *Power*.

4.1 MORE ABOUT INHERITANCE

Above I have dealt with the most important concerning inheritance, but there are some details back. The kind of inheritance, as discussed above, is called for *linear inheritance*, which means that a class can only inherit one class – a class can have only one base class. On the other hand, a class can have all the derived classes, as may be needed, and you can inherit in all the levels as you wish. Inheritance leads to a hierarchy of classes. As an example the classes concerning loans has the following hierarchy:



ILoan is an interface, but it is included in the type hierarchy in the same manner as classes. As another example, consider class hierarchy from problem 2:



If a class can inherit several classes, wee talk about multiple inheritance, but Java supports only linear inheritance. Java does indirectly support multiple inheritance when a class can implement all the interfaces that are needed. It is not inheritance, but it means that at design time where you decide which classes a program must consist of, can work with multiple inheritance. You can design classes that implements multiple interfaces.

5 THE CLASS OBJECT

Java has a class called *Object*, and all classes inherits directly or indirectly this class. If, as an example looking at the class *Loan*

```
public abstract class Loan implements ILoan
```

the class automatically inherits the class *Object*. It means that you could have written

```
public abstract class Loan extends Object implements ILoan
```

and indeed it is allowed to write – but unnecessary. This means several things, including as perhaps the most important that all classes in Java are linked in a large hierarchy with the class *Object* as the root. All classes without exception is an *Object* and inherits the properties as an *Object* has.

The class *Object* defines a few basic methods that all has a trivial implementation, and it is then up to the individual classes to override the methods so that they get a reasonable use. Below I briefly mention the most important methods, and I will use the classes from the project *Students*. The class *Subject* is defined in the following way

```
public class Subject implements ISubject , IPoint
```

where its methods are defined in two interfaces. The class also has a *toString()* method

```
public String toString()
{
 return navn;
}
```

which returns the subject's name as a *String*. If you now remove this method (comment it out) and perform the following method:

```
private static void test01()
{
 try
 {
  ISubject subject = Factory.createSubject("MAT7", "Matematics");
  System.out.println(subject);
  subject.setName("Matematics 7");
  System.out.println(subject);
 }
```

```
 catch (Exception ex)
 {
  System.out.println(ex.getMessage());
 }
}
```

you get the result:

```
students.Subtect@6d06d69c
students.Subject@6d06d69c
```

subject is an object, and when *System.out.println()* prints an object, it is the result of the object's *toString()* method that is printed. If now the object's class does not have a *toString()* method – such as the class *Subject* – it is *toString()* from the base class that is performed, and in this case it is *toString()* from the class *Object*. The class *Object* has a default *toString()* method that prints the class's name followed by a number. The number is the reference to the object on the heap specified as a hexadecimal number. Of course it makes no particular sense, but it means that all objects can be printed with *System.out.println()*, or more precisely that one has a representation of any object as a text. It is then up to the programmer of the object's class to override the *toString()* so that it returns a reasonable result. I will immediately remove the comments for *toString()* method in the class *Subject* again:

```
Matematics
Matematics 7
```

If you in the above method writes the statement

```
String s = "The subject: " + subject;
```

the result will be that the variable *s* has the value

```
Subject: Matematics 7
```

The reason is that the plus operator is interpreted as string concatenation, and since *subject* is an object, it is the value of this object's *toString()*, that is concateneted. Incidentally, the result would be the same if you wrote:

```
String s = "The subject: " + subject.toString();
```

Another method in the class *Object* is called *getClass()*, and it returns an object of the type *Class* which contains information about an object's class – and there are many. Consider as an example the following method:

```
private static void test02()
{
 try
 {
  ISubject subject = Factory.createSubject("MAT7", "Matematics");
  print(subject);
 }
 catch (Exception ex)
 {
```

```
   System.out.println(ex.getMessage());
  }
 }

private static void print(Object obj)
{
 System.out.println(obj.getClass().getName());
 for (Method m : obj.getClass().getMethods()) System.out.println(m.getName());
}
```

Note that the method *print()* use a class *Method*, that is defined in the package

```
java.lang.reflect
```

The method *print()* has a parameter of the type *Object*, and for this object it prints the name of the object's class, and the names of the object's methods. The method *test02()* creates an object of the type *Subject* and sends it as a parameter to the *print()* method. The result is the following:

```
students.Subject
getECTS
setECTS
setName
getName
toString
getId
wait
wait
wait
equals
hashCode
getClass
notify
notifyAll
```

You should note the class name, and the object is a *Subject*, although in the method *test02()* is known as an *ISubject*. You should also note that the metod *print()* prints the names of total 14 methods. It's all public methods, and it is far more than the class *Subject* defines, but the rest comes from the class *Object* – that you know is the base class for *Subject*.

A *Class* object has many methods, so you can get many details regarding an object. The method *getClass()* can not be overridden.

The class *Object* also has a method called *equals()* which has a parameter of the type *Object* and is used to test whether two objects are the same. Consider then the following method:

```
private static void test03()
{
 try
 {
  ISubject subject1 = Factory.createSubject("MAT7", "Matematics");
  ISubject subject2 = Factory.createSubject("MAT7", "Matematics");
  System.out.println(subject1.equals(subject2));
 }
 catch (Exception ex)
 {
  System.out.println(ex.getMessage());
 }
}
```

The method creates two objects of the type *Subject* and tests whether they are the same. If you executes the method, you get the result

```
false
```

which is not really what one would expect. The method *equals()* is defined in the class *Object*, and works by comparing the two objects addresses and thus whether they refer to the same object on the heap. In this case refers *subject1* and *subject2* to different objects, and therefore returns *equals() false*. Now if you want the comparison instead to be done by from the objects' values or state, it is up to the programmer to override the method in the concrete class.

If you look at the class *Subject*, I would say that two *Subject* objects are equal if they have the same id – the value is perceived as a key that can identify a *Subject*. You can then override *equals()* in the following way:

```
public boolean equals(Object obj)
{
 if (obj == null) return false;
 if (getClass() == obj.getClass())
  return id.toUpperCase().equals(((Subject)obj).id.toUpperCase());
 return false;
}
```

That is, a *Subject* object is equal to another object, if the other object is not null, and if the other object also is a *Subject*, and if both of these *Subject* objects have the same id when there is no distinction between uppercase and lowercase letters. You should note that the test build on, that the class *String* overrides *equals()* such i compare the values af strings. You should also note that there is no distinction on the name or ECTS, and that it is the programmer's decision how the *equals()* must be overridden and there could be other options. If you now executes the method *test03()* you get result

```
true
```

A method, which belongs to *equals()* is *hashCode()*. The method returns an *int*, and the default implementation in class *Object* returns the object's memory address. The method is used to identify an object, as well as to represent an object as a number. Later I will show applications of the method *hashCode()*, and here I will also explaine how to overrides this method, but in principle you should always override *hashCode()* if you overrides *equals()*. If need be the protocol is that if two objects are equals, they must have the same hash code. Because the class *String* overrides *hashCode()* – correctly – it can be overridden in the class *Subject* as follows:

```
public int hashCode()
{
 return id.toUpperCase().hashCode();
}
```

The class *Object* also has a method with the following signature:

```
protected void finalize() throws Throwable
```

Because it is protected, it can not be called from other classes, but it can be overridden. The method has not so many uses, and the default implementation performs nothing, but the method is called by the garbage collector when an object is removed from the heap. You has the possibility, to override the method to add code to be executed when an object is removed from the heap.

The last method in class *Object*, which I will mention has the following signature:

```
protected Object clone() throws CloneNotSupportedException
```

and is again a method which you not the immediately can call from another class. The idea is that the method should create a copy of the current object. A class can override this method if it implements the interface *Cloneable*. The default implementation tests whether the object's class implements this interface, and if not the method raises an exception.

If I extends the definition of the class *Subject* as follows

```
public class Subject implements ISubject, IPoint, Cloneable
```

the default *clone()* method will create an object that is a copy of the current object and hence have variables with the same value as the current object. In many contexts it is ok, but if you want to determine how the copy is created you can be override *clone()*. If, for example the object has references to other objects, it may be necessary, and another reason may be that the method in *Object* is protected. In the class *Subject* you could override the method to something like the following:

```
public Object clone() throws CloneNotSupportedException
{
 Subject subject = null;
 try
 {
  subject = new Subject(new String(id), new String(name), ects);
 }
 catch (Exception ex)
 {
 }
 return subject;
}
```

That the creation of a *Subject* object is wrapped in try/catch it is because I have written the constructor of the class *Subject* so that it can raise an exception. You should also note that I as actual parameters to the constructor creates copies of the strings. Now it is not really necessary, but is included because it in other contexts, in the case of other class types than *String*, is often necessary.

You should also note that the method is *public* although in *Object* it is *protected*. It is actually allowed on that way to strengthen the visibility of a method such that in a derived class is defined with greater visibility than in the base class. One can, in turn, not reduce visibility.

Consider the following method:

```
private static void test04()
{
 try
 {
  ISubject subject1 = Factory.createSubject("MAT7", "Matematics");
  ISubject subject2 = (Subject)((Subject)subject1).clone();
  System.out.println(subject1.equals(subject2));
  System.out.println(subject1 == subject2);
 }
```

```
  catch (Exception ex)
  {
   System.out.println(ex.getMessage());
  }
 }
```

If you execute the method, you get the result:

```
true
false
```

because the two objects is indeed a copy of each other and, therefore *equals()*. The last statement prints *false* and it shows that *subject1* and *subject2* are two different objects. You should note the statement

```
ISubject subject2 = (Subject)((Subject)subject1).clone();
```

subject1 is in the code known as an object of the type *ISubject*, and it does not have a *clone()* method. Therefore, it is necessary with a type of cast of *subject1* to a *Subject*. The method *clone()* returns an *Object*, and therefore it is necessary with a typecast of the return value. The concept typecast is elaborated in the next chapter.

6 TYPECAST OF OBJECTS

I have previously in Java 1 mentioned typecast of simple types. As an example you can not explicit copy a *long* to an *int* because an *int* may not have enough space. It may be necessary to write something like the following to tell the compiler that *t* well should be copied to *a*:

```
long t = 123;
int a;
a = (int)t;
```

Also objects can be typecasted, but it is necessary that they belong to the same class hierarchy. Otherwise, you get an exception if not the compiler before is giving an error. Consider as an example the following method:

```
private static void test05()
{
 try
 {
  Subject subject1 = new Subject("MAT7", "Matematics");
  ISubject subject2 = subject1;
  Subject subject3 = (Subject)subject2;
  System.out.println(subject1.equals(subject2));
  System.out.println(subject1.equals(subject3));
  System.out.println(subject3.equals(subject2));
  // Course course = (Course)subject1;
 }
 catch (Exception ex)
 {
  System.out.println(ex.getMessage());
 }
}
```

Here refers *subject1*, *subject2* and *subject3* all to the same object. Note that you can write

```
subjet2 = subjet1
```

for *subject1* is a *Subject* object and thus specifically an *ISubject* object. In contrast, the statment

```
subjet3 = (Subjet)subjet2
```

requires a typecast, as *subject2* is an *ISubject* object, which is not necessarily a *Subject* object. Note the syntax of a typecast, where in front of a variable you write the type to cast to, in parentheses.

By contrast, the following statement

```
Course course = (Course)subject1;
```

results in a compiler error because *Course* and *Subject* not are types from the same class hierarchy.

7 A LAST NOTE ABOUT CLASSES

As a final note regarding classes I mention the word *final*. If one defines a variable *final* like *NAME* in the class *Institution*

```
public final static String NAME = "The Linux University";
```

this means that the variable is a constant, and its value can not be changed. Also methods can be defined *final*, and a *final* method can not be overridden in a derived class. If a method is overridden in a derived class, you has the possibility to write the method to have a whole different meaning, and if you do not want it to be possible, you can define the method *final*. Also *final* classes can be defined, and a *final* class is a class which can not be inherited. As an example is the *String* class final.

7.1 CONSIDERATIONS ABOUT INHERITANCE

I will conclude all what is said about interfaces and inheritance with a few remarks. Seen from the programmer consists a program of classes that defines the objects used by the application to perform its work. For the sake of maintenance of the program you are interested that these classes is as loosely coupled as possible, and that is, that the classes should know so little about each others as possible. In that way you can change the classes without it affects the rest of the program. You should therefore strive to write the classes, so a change only affects the class itself and thus only has local significance.

Now you can not write programs without there are couplings between the classes, the meaning of a class is precisely that it must make services available for others to use, but you can ensure that the coupling only takes place through public methods. That's why I sometimes have talked about data encapsulation where instance variables are kept private, and thus it becomes the programmer who through the class's public methods decides which couplings to be possible. This principle can be further supported through an interface if all the classes are defined by an interface. An interface defines through abstract methods (methods in an interface are by default public abstract), which couplings should be possible, and adhere to it and always define references to objects using a defining interface, you get a looser coupling of the program's components as an object alone is known as an interface and the class that implements the interface is not known and may be changed without it affects the rest of the program. Therefore, it is a design principle that that you should program to an interface.

To program to an interface is a good design principle, but everything can be overstated. A program can easily consist of several hundred classes, and if each (or most) of these classes defines an interface, the number of types will be huge and may lead to code that is difficult to grasp. One should therefore consider, when a class must define an interface and concerning primary classes, which represent key concepts in which there is a chance that there will be changes. A program will always consist of many classes that are using other classes that are local in nature, and these classes should not be defined with an interface. The upshot is that programming to an interface is an important principle and one of the best ways to secure loose couplings between parts of the program, but also that, while developing consider whether an interface makes sense.

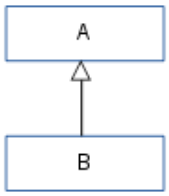
Another use of interfaces is that using interfaces at design time you can define different contracts that the program's objects must comply. As a class can implement multiple interfaces (all of them you need), you can create objects that meet multiple contracts, and as some of the most important thing you can test whether an object meets one or the other contract. With reference to the previous chapters, an object defined *Cloneable* abide by the contract, that it can create a copy of itself. Another interface is called *Comparable*, and classes that implements this interface, instantiates objects that can be arranged in order and hence, for example be sorted. Java defines a number of such interfaces, which defines one or another property that an object may have. You will later in the book Java 4 about collection classes face classes that implements many interfaces. An interface is a concept that defines contracts which objects must comply.

If you look at the interfaces defined in this book, they reflect very closely the classes that implements these interfaces. It does absolutely not necessarily be the case, and many interfaces define a contract in form of a few and perhaps only a single method.

With regard of inheritance one can say that there are two uses. In one situation you have some classes which are similar, such that they somewhat are comprised of the same variables and methods. In such a situation you can take what the classes have in common, and move it into a common base class, that the others inherits. That way, you avoid the same code can be found in several places and thus have a program that takes up less space, but the important thing is that if you have to change the code, you should only change it at one place, and you are sure that you do not forget to change somewhere. The second situation is that you have defined a concept in the form of a class, and then you need a different concept, similar to the first but might need an extra variable or method, or there may be a method that should work on another way. In this case, the new concept can be defined as a class that inherits the first class. The two situations are really two sides of the same coin, but in the first case we speak of generalization, while in the second case, wee talk about specialization. Whatever's inheritance reflects the fact that a program consists of classes that have something in common and to model concepts of the same kind, and may be useful to define a hierarchy of classes that inherit each other.

At the design level the implementation of an interface also is a form of inheritance as an interface says that an object satisfies a particular contract. Because Java only supports linear inheritance, interfaces help during the design to work with multiple inheritance.

The most important of inheritance is not so much the expansion of existing code (although it is of course also important), but it is the relationship between super class and subclass. If you have a specialization

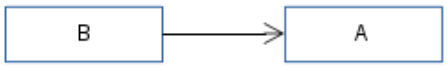


(and here it is not so important, if A is a class or an interface), it is important that a *B* is also an *A* and a *B* anywhere can be treated in the same way as an *A*. It means that you can write a method and thus code which treats an *A* object:

```
type metode(A a)
{
}
```

and the method will work regardless of whether the parameter is an *A* object or *B* object or another type that directly or indirectly inherits *A*. The method does not know the specific type, and it does not have to do it. It is called *polymorphism* and is one of the most important concepts in object-oriented programming and allows you to write program code that is very flexible and general.

Inheritance is one of the basic principles behind object-oriented programming and more than that, for it is a requirement that an object-oriented programming language supports inheritance. However, it is not the solution to everything, and there is also criticism of inheritance. Taking the above design where *B* inherits *A*, it reflects a strong connection and it especially if *A* defines *protected* members. There is a high probability that changes in *A* will have an impact on derived classes. Moreover, one can mention that the connection between *A* and *B* is very static and determined at compile-time. Sometimes one thinks, therefore, on a design as



wherein an object *B* knows an *A* object through a reference. Here reference can be replaced at runtime, providing the opportunity to the object *B* that it can refer to another *A* object, and the result is a code that is more flexible. One speaks sometimes that *B* delegate tasks to the *A* object. The two design principles are not directly each other alternatives, but in the last years there has probably been briefed on using delegation in situations where you would previously use of inheritance.

PROBLEM 3

In this task, you must write some classes that are organized in a hierarchy. The classes are simple and represent geometric objects, triangles and squares, and to the extent that there is a need for mathematical formulas, it is simple formulas, which are known from primary school. The goal is to show many of the concepts of classes treated in this book.

Start by creating a NetBeans project, you can call *Geometry*. The example treates as mentioned geometric shapes, but you should only be interested in a shape's perimeter and area. Correspondingly, a shape can be defined as follows:

```
package geometry;

/**
 * Defines a geometrical shape.
 */
public interface IShape
{
 /**
  * @return The circumference of the shape
  */
 public double perimeter();

 /**
  * @return The area of the shape
  */
 public double area();
}
```

and you need to add this interface for your project. You must then add an auxiliary class that should be called *Point* and represents a point in a plane coordinate system:

```
package geometry;

public class Point
{
 public static final double ZERO = 1.0e-20; // represents zero

 private double x;
 private double y;
 public Point(double x, double y)
 {
  this.x = x;
  this.y = y;
 }
```

```
public double getX()
{
 return x;
}

public double getY()
{
 return y;
}

public double length(Point p)
{
 return Math.sqrt(sqr(x – p.x) + sqr(y – p.y));
}

public String toString()
{
 return String.format("(%.4f, %.4f)", x, y);
}
```

```
 public boolean equals(Object obj)
 {
  if (obj == null) return false;
  if (getClass() == obj.getClass())
  {
   Point p = (Point)obj;
   return Math.abs(x - p.x) <= ZERO && Math.abs(y - p.y) <= ZERO;
  }
  return false;
 }

 public static double sqr(double x)
 {
  return x * x;
 }
}
```

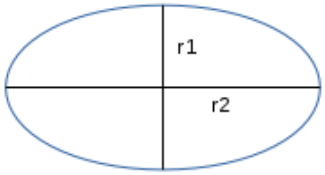
Regarding the method *length()*, note that given two points (x_1, y_2) and (x_2, y_2) you determines the distance between the points as:

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Also note that the class has a static method *sqr()* which returns the square of a number. In principle, such a method does not have to do with a point and is also just an auxiliary method to implement *length()*, but because it may be useful in other contexts, it is defined as a *static public* method.

Write an abstract class *Shape* that implements the interface in *IShape*. The class should only overrides the method *equals()*, such two shapes are the same, if the objects has the same class, and if the two shapes have the same perimeter and area.

The next class will represent a concrete geometric shape. The class should be named *Ellipse* and must represents an ellipse, when an ellipse is solely defined by the two radi that are transferred as parameters to the constructor:



You can calculate the ellipse's perimeter and area on the basis of the following formulas:

$$area = r_1 r_2 \pi$$
$$perimeter = 2\pi\sqrt{0.5(r_1^2 + r_2^2)}$$

With the ellipse in place, you can write a class *Circle*, representing a circle when a circle is simply an ellipse where the two radi are equal. The class *Circle* can be written as a class that inherits *Ellipse*.

As a next step, yot should write classes to triangles. Start with a class *Polygon*, which can represent an arbitrary polygon. It can be represented by *Point* objects that are the polygon's vertices. The class should not validate the corners and test whether they lead to irregular polygons, for example polygons where sides intersect. A polygon can be written as the following class:

```
package geometry;

public abstract class Polygon extends Figur
{
 protected Punkt[] p;  // polygonens hjørner

 public Polygon(Punkt … p)
 {
  this.p = p;
 }
 public double omkreds()
 {
  double sum = p[p.length – 1].length(p[0]);
  for (int i = 1; i < p.length; ++i) sum += p[i – 1].length(p[i]);
  return sum;
 }

 public String toString()
 {
  …
 }
}
```

Knowing the vertices, you can immediately implements the method *perimeter()* as the circumference can be determined as the sum of the lengths of the edges. However, you can not immediately implement the method *area()*, and the class is therefore defined *abstract*.

Next, write a class *Triangle*, which must be a class that inherits *Polygon*. The class must have a constructor that has three points as parameters, and apart from the constructor the class alone has to override the *toString()* and implement the method *area()*. Regarding the last you can determine the area of a triangle with sides a, b and c as follows:

$$areal = ½ab\sqrt{1 - \left(\frac{a^2 + b^2 - c^2}{2ab}\right)^2}$$

In the interest of other classes the area calculation should be done by means of using a *public static* method that has as parameters has the triangle's vertices.

The class *Triangle* represents an arbitrary triangle defined by three points. You can also have more specialized classes for the triangles, such as *Equilateral* class representing an equilateral triangle, which should be a class that inherits the class *Triangle*. The class's constructor must have one parameter that is the side length (note that it is easy to define three points which form the vertices in a triangle with a certain side length – start with the point (0,0)). For performance reasons the class should override the method *perimeter()*, since it can be implemented more effectively if you know the side length. Write the class *Equilateral*.

Write similar to a class *RightAngled* that represents a right angled triangle.

As the last shapes you should write some classes to squares. Start with a class *GeneralSquare*, which inherits the class *Polygon*. The class must have a constructor, which parameters are four *Point* objects. Furthermore the the class must implements the method *area()*, which is abstract in the base class. There is no general formula to determine the area of a general square, but you draw a diagonal, that divides the square into two triangles (because I ignore the problem that a square may be concave). You can therefore implement the method *area()* by determining the area of the two triangles.

You must then write a class *Rectangle*, which inherits the class *GeneralSquare* when the class's constructor has two parameters that respectively are the width and height. Also, write a class *Square* when a square just is a rectangle where the sides are of equal length.

You've written some classes that represents geometric objects, and you must now define a composite shape, which consists of other shapes. The following interface inherits *IShape* and defines a composite shape called a *IDrawing*:

```
package geometri;

public interface IDrawing extends IShape
{
 public void add(IShape … shape) throws Exception;
}
```

Write a class *Drawing* that implements this interface when the circumference of a drawing is defined as the sum of the perimeters of all the drawing's shapes and the same for the area.

A drawing is therefore especially an *IShape*, and thus a drawing contains other drawings. Below is a test program that creates a drawing that consists of

1. a drawing that contains a circle, a equilateral and a rectangle
2. a right angled triangle
3. a drawing that contains a square and two circles

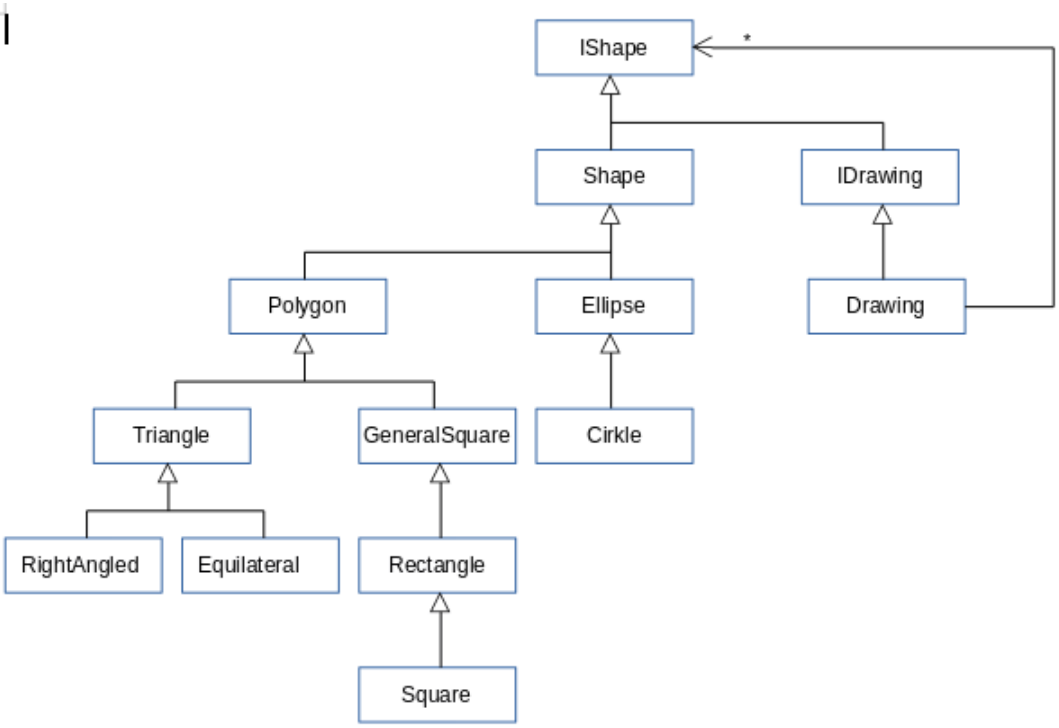
```
package geometry;

public class Geometry
{
 public static void main(String[] args)
 {
```

```
 print(create(create(new Circle(2), new Equilateral(3), new Rectangle(3, 4)),
  new RightAngled(3, 4), create(new Square(5), new Circle(1), new Circle(3))));
}

private static void print(IShape shape)
{
 System.out.println(shape);
 System.out.println("Perimeter: " + shape.perimeter());
 System.out.println("Area:    " + shape.area());
}

private static IShape create(IShape … shapes)
{
 IDrawing d = new Drawing();
 try
 {
  d.add(shapes);
 }
 catch (Exception ex)
 {
  System.out.println(ex.getMessage());
 }
 return d;
}
```

Test the program. The result should be something like the following:

```
Start drawing
Start drawing
Circle, r = 2,00000000
Equilateral: 3
Rectangle: width = 3.0, height = 4.0
End drawing
Right angled triangle with kateters: 3.0 and 4.0
Start drawing
Square, side 5.0
Circle, r = 1,00000000
Circle, r = 3,00000000
End drawing
End drawing
Perimeter: 92.69911184307752
Area:    90.87941146728707
```

The types of this task is organized in a class hierarchy and can be illustrated as shown below. It's simple to expand the class hierarchy with new classes, including classes for new shapes, and as long as these classes complies with the contract *IShape*, the class *Drawing* can use these classes as it knows nothing about the concrete classes.

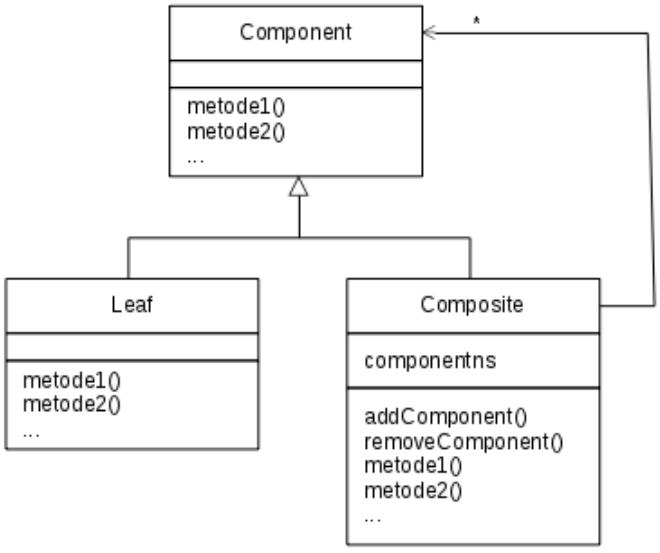

IShape
*
Shape
IDrawing
Polygon
Ellipse
Drawing
Triangle
GeneralSquare
Cirkle
RightAngled
Equilateral
Rectangle
Square

7.2 THE COMPOSITE PATTERN

In fact it is quite often that, in practice, you meat a situation similar to above and think for example on a part list where you have a product that is composed of subcomponents, or think of a menu where a menu consisting of menu items or other menus. Another example is a filesystem consisting of directories and files. Therefore, it is natural to express the situation in a design pattern called a *composite pattern*.

The problem is thus to manipulate a hierarchy of objects, where the bottom of the hierarchy have some concrete objects that we call the leaf objects, and there are some other objects that we call composite objects consisting of leaf objects and other composite objects. Both leaf objects and composite objects have a common base class (and possibly it is an interface or an abstract class), and the idea is that composite objects treat their child objects alike, whether in the case of leaf or composite objects. The solution is illustrated with a figure as shown below and the solution is referred as a *composite pattern*.

In practice, there may be different flavors, and often there may be several kinds of leaf objects that all are part of a hierarchy. This is the case in the above example with geometric shapes. Here, the *Component* class corresponds to the interface *IShape*, and the class *Leaf* corresponding to *Shape*. In the figure below corresponds the class *Composite* to *IDrawing*. In principle, it is a very simple pattern, but in practice there are still some considerations on how to implements the pattern. The goal is as mentioned that the *Leaf* and the *Composite* objects must have the same behavior, and therefore the methods *addComponent()* and *removeComponent()* is defined in *Component*. It provides, however, a problem since they do not make sense for *Leaf* objects. Where necessary, they must in *Leaf* classes be implemented as methods that do not perform anything and possible raises an exception. If you instead uses a design as shown above, it then is necessary to test whether specific objects are *Leaf* or *Composite* objects.



8 FINAL EXAMPLE

As a conclusion of this book, I will write a program that can evaluate matheatical expressions. The user should be able to enter a mathematical expression that depend on one or more variables, for example:

$$2 * \sin(x0) + sqrt(5 * x1 + 3)$$

that depends on two variables. Then the user should enter values for the expression's variables and the program should then evaluate the expression for these variables.

If there is an error:

- the user has entered an expression that is not syntactically correct
- the user has entered arguments that are not legal compared to the current expression
- an error occurs when the expression is evaluated

the program must show an appropriate error message.

In the example above the expression includes sine and square root. The program should support the most common mathematical functions somewhat similar to what applies for a mathematical calculator, and it is a desire that it should be easy to extend the program with new functions, if the need arises.

The program must have a graphical user interface where you can enter expressions and values for the variables. With regard to expressions there are following requirements:

- An expression is not case sensitive, and it should not matter whether you write in lowercase or uppercase.
- An expression may contain any number of variables, referred to as x0, x1, x2, …, that is an x followed by a non-negative integer or number.
- Numbers is always entered with dot as decimal point.
- An expression must support the four common arithmetic operators +, -, * and /.
- It should be allowed to use parentheses in any number of levels.
- If a mathematical function has several arguments, they must be separated by comma.

8.1 ANALYSE

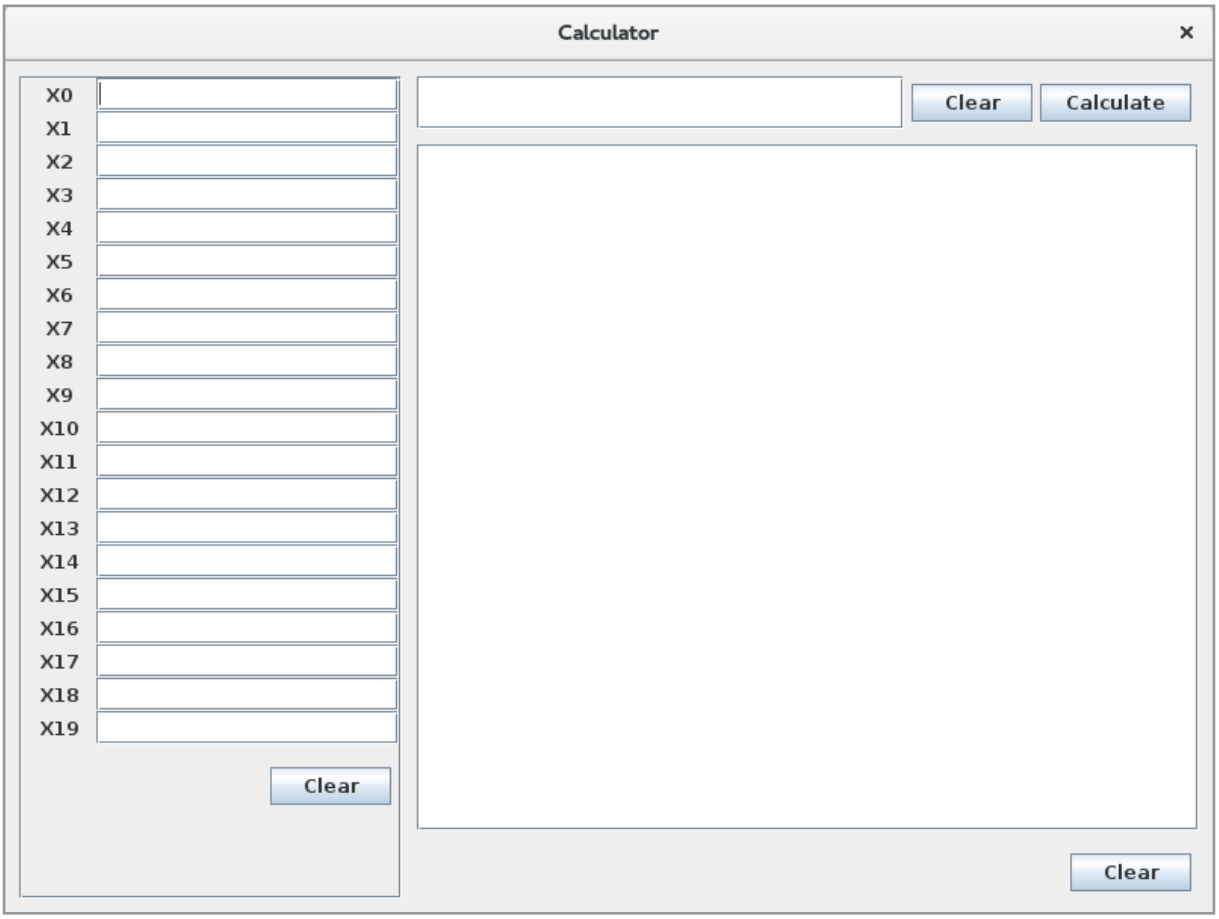
The above can be seen as the requirements for the program. It is always the starting point for a sofware development project, but before you can address the development of the program it is the typical needed to clarify uncertainties or boundaries. This is also true in this case where it is necessary to clarify which functions the program exactly must support and how the user interface should be.

A mathematical function is identified by a name and then it can be followed by a parameter list in parentheses. In an expression, it must be possible to use the following functions:

- constant functions (functions without parentheses)
  - pi
  - e
- functions in 1 variable
  - sin cot sqr
  - asin acot sqrt
  - cos ln abs
  - acos exp frac
  - tan log floor
  - atan antilog

- functions in 2 variables
  - pow
  - root

To define the application user interface I has created a NetBeans project named *Calc*. The project creates a prototype, which is a simple application that opens the following window:

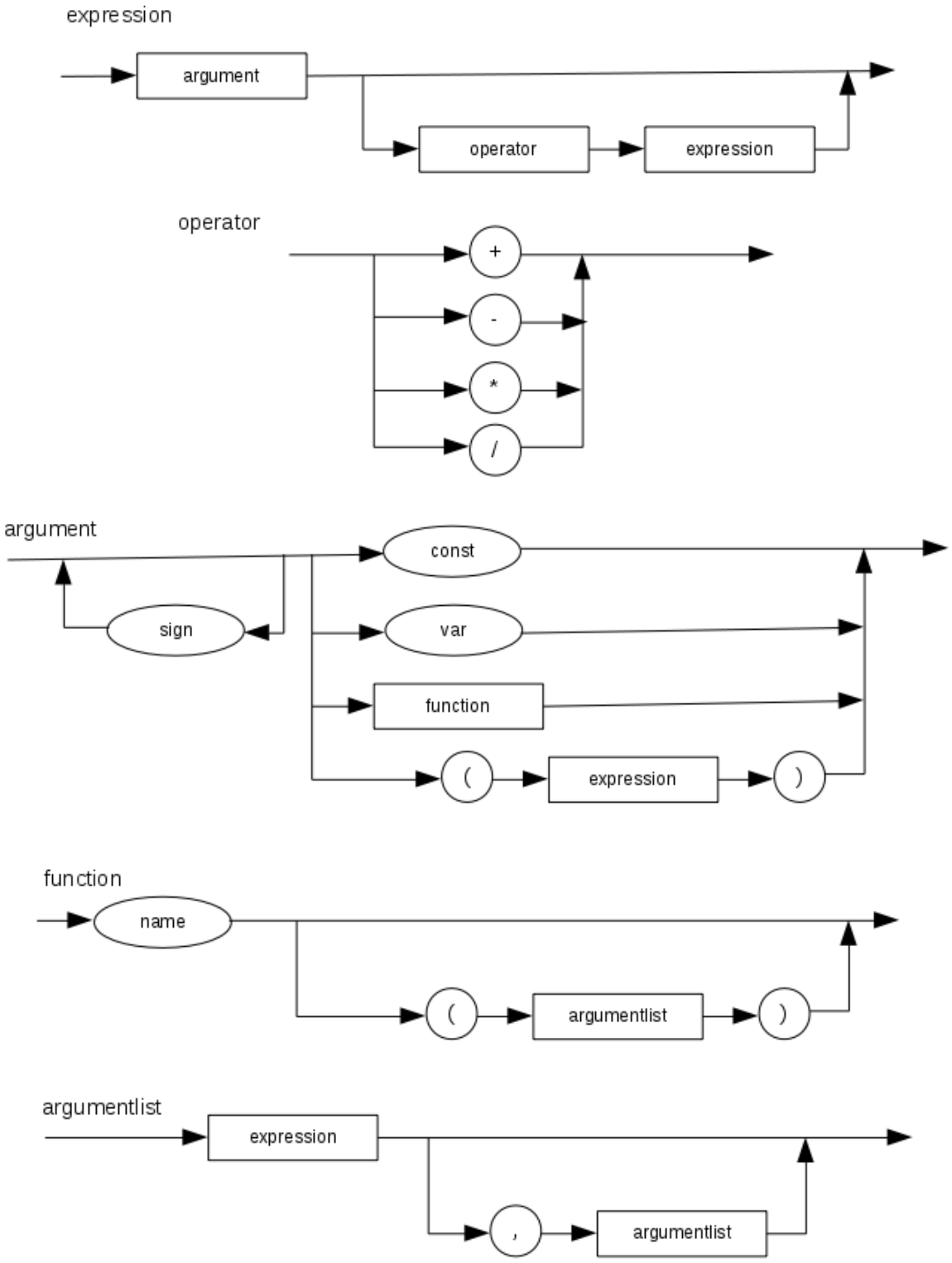


That it is a prototype only means that it is a program that is not doing anything. The buttons have no function, but it can be a good place to start the development of a program to clarify what it's all about. First, it is relatively quickly to develop such a program, and partly to the prototype can be presented to the future users, where it can be a useful tool to clarify that the task is understood correct.

In this case it is decided, that the program should be able to work with 20 variables, that the program must have an input field for entering an expression and a list box for the results.

AN EXPRESSION

Exactly an expression can be described with the following syntax diagrams:



An argument can start with one or more sign characters (plus or minus). Then there are four options

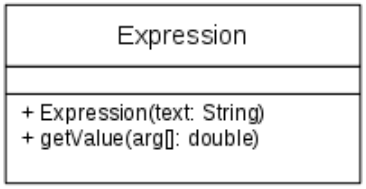
1. *const*, that always must be a non-negative number – a string that can be converted to a *double*
2. *var*, that is a variable and then a string at format *Xn*, where *n* is an index
3. *function*, that is a mathematical function
4. an expression in parentheses

What remains is to define what a function is, and it appears from the above.

The value of an expression must always be a *double*.

8.2 DESIGN

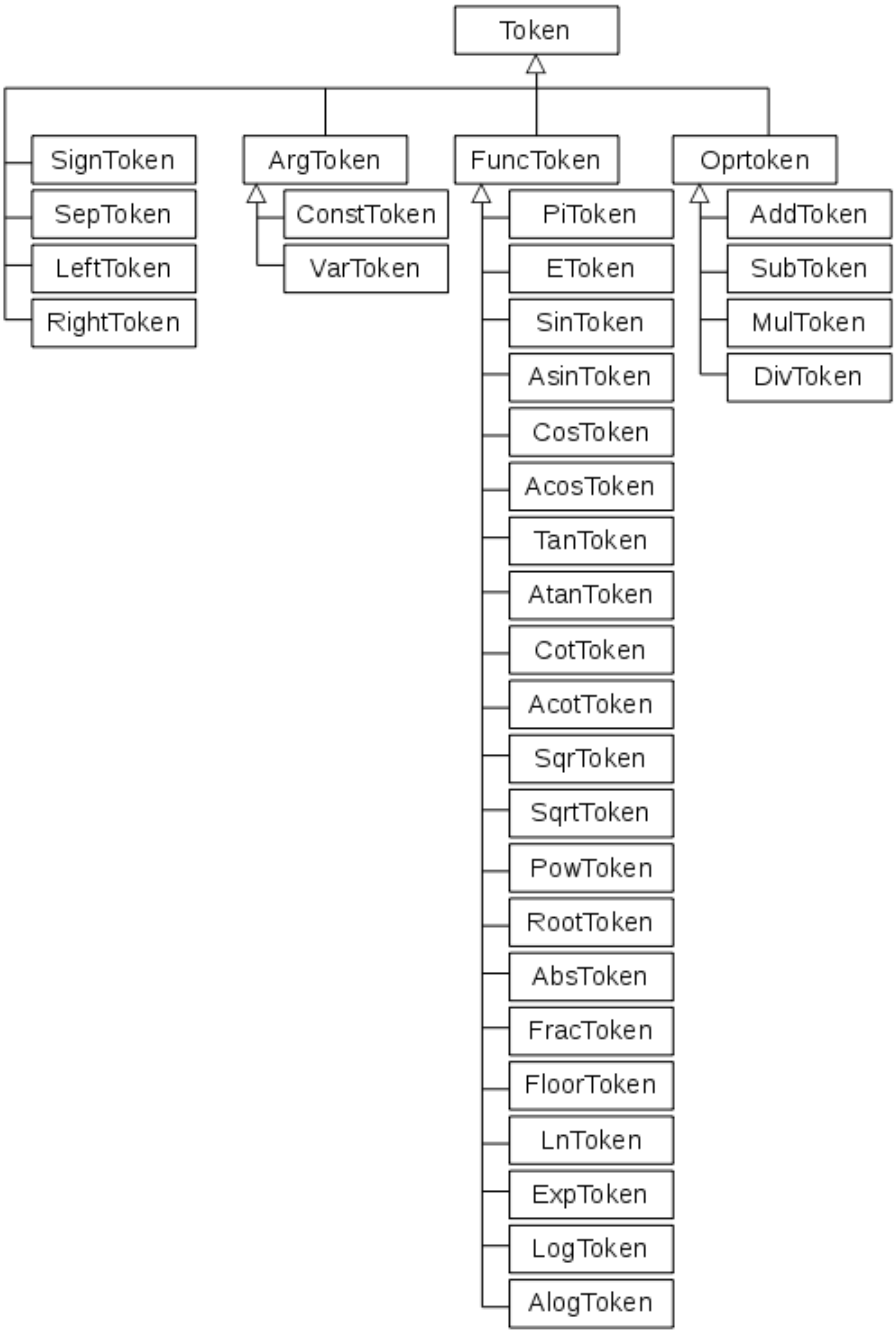
An expression must be represented by a class named *Expression*. In principle, it is a very simple class, that in addition to a constructor simply consists of a single method that returns from arguments the expression's value:



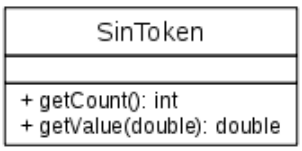
Here it is the constructor, that is complex since, as parameter it has the expression as a string, and it is accordingly the constructor's task to split the string up into tokens and validate whether these tokens represents a legal expression. A token is an item that can be included in an expression, and a token is a substring such as cos, + and a number. According to the above syntax diagrams and associated comments there are following tokens

+	addition or sign	-	subtraction or sign	*	multiplication
/	division	(	left parenthesis	)	right parenthesis
,	argument separator	sin	sinus	asin	arc sinus
cos	cosinus	acos	arc cosinus	tan	tangens
atan	arc tanges	cot	cotangens	acot	arc cotangens
ln	natural logarithm	exp	exponential function	log	logarithm 10
alog	antilog	sqr	square function	sqrt	square root
pow	power of	root	root of	abs	absolut value
frac	fraction	floor	interger part	pi	the constant pi
e	the constant e				
xn	a variable that start with x and is follwed en non-negative integer				
num	a number that is a non-negative decimal number with . as decimal point				

This can result in 30 different tokens – and indeed 31 as plus or minus especially can be a sign. The different tokens has to be treated different, but can be arranged into groups with common properties, and as such, I will define the following class hierarchy, where each token is defined by a class:



It is very simple classes, and as an example is *SinToken* a class that represents a sine function. The class is derived from *FuncToken*, and the class has only two properties, where the first returns the number of arguments, while the second determines the function's value from a single argument:



The constructor in the class *Expression* must perform three operations:

- Scanning the expression which means that the string should be split into the tokens that the expression consists of.
- Parsing the expression that means to control that the expression's syntax is correct according to the syntax diagrams.
- Converting the expression to postfix form, meaning that the expression's elements must be reorganized into a sequence of tokens in postfix form.

To solve the first problem, the parameter string is divided into tokens of the kind defined above. When the string is divided you have a number of tokens, each of which is a string, and all these strings must be converted to a *Token* object.

After the scanning, you have a list of tokens, and parsing has to investigate whether this list of tokens is in accordance with the syntax rules. It is not simple, but it means to write a method corresponding to each of the above syntax diagrams.

A STACK

In the book Java 1 I described an *ArrayList* as an example on a collection class, and I have used an *ArrayList* many times since. In the first chapter in this book I mentioned the program stack as a data structure, where the runtime system store parameters and local variables. Java also implements a collection class as a stack, and I need that class in the following, and therefore little about what a stack is.

It is as an *ArrayList* a container, that can store objects of a particular type of, but it is a very simple data structure with only two importen operations:

1. *push* that put an object on the stack (store a value at the top of the stack)
2. *pop* that returns an remove the object on the top of the stack

It is as such a LIFO data structure, where the object that is removed from a stack is the last object, that is put on the stack. In Java a stack also has other operation, and as an exampel and operation *peek*, that returns the object on the top of the stack but without remove it. It means that *peek* only look at the top of the stack.

As an *ArrayList* there is no limit om then number of objects a stack can contains.

In Java the type is called *Stack*, and I uses the type in the program, bul I vil also refer to a stack below, and therefore this remark.

TO POSTFIX

Usually you write an expression on infix form which means that you writes the operator between two operands, as for example:

2 + 3

which means the sum of 2 and 3. If there are several operators, we need rules for how the expression must be evaluated. For example means

```
2 * 3 + 4
```

that you first calculate the product of 2 and 3 and then adds this result to 4 – the result is 10. In contrast, means

```
2 + 3 * 4
```

that you first calculates the product of 3 and 4, since multiplication has higher precedence than addition – the result is therefore 14. If you wish to suppress this rule, you has to use parentheses:

```
(2+3) * 4
```

and the expression has the value of 20.

If in an expression there are several operators of equal priority, the rule is that the operators are evaluated from left. Below is first computed the sum of 2 and 3 and then subtract 4 because addition and subtraction have the same priority:

```
2+3−4
```

An expression may be more complex, for example

```
(1+2 * (3+4))(/(5+6) * 7)
```

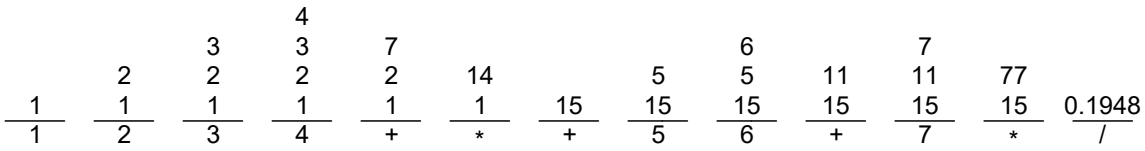
where there are parentheses within the parentheses. The value of the expression is 0.194805. When an expression contains parentheses, the parentheses are evaluated first, starting with the innermost parentheses. It is certainly possible to write a method that does it, but if the expression becomes more complex with mathematical functions and many parentheses, it is not simple, and therefore one will typically convert the expression to postfix form. This means that an operator is written after the operands. Thus, the above expression's is written as

```
2 3 +
2 3 * 4 +
2 3 4 * +
2 3 + 4 *
2 3 + 4 –
1 2 3 4 + * + 5 6 + 7 * /
```

If for example you must calculate the value , you have a list of five tokens, and you evaluates the expression by traversing the list from left to right, and every time you encounters an operator it acts on the two operands preceding:

2 3 * 4 +
6 4 +
10

The idea is that any expression can be written in postfix form without using of parentheses. When the expression should be evaluated, it is traversed just from left to right. Every time you come to an operand, put it on a stack. Is it an operator you pop the stack twice (if it is an operator with two arguments) calculates the result and put it on the stack. Finally the stack will contain only one element which is the result. The method may based on the last of the above expressions be is illustrated in the following manner:



The conclusion is that it is much easier to evaluate an expression in postfix form than one in infix form and it is therefore worthwhile to seek a strategy (an algorithm) to convert an expression from infix to postfix form. It may be done in the following manner by using a stack:

```
the expresseion is traversed from left and for every tooken
       1.  if it is a sign push it on the stack
       2.  if it is a function push it on the stack
       3.  if it is a variable push it on the stack
       4.  if it is a number push it on the stack
       5.  if it is a left parenthes push it on the stack
       6.  if it is a right parenthes, then pop the stack and add the top
       of the stack to the resultat until you get a left parenthes
       7.  if it is an operator then pop the stack and add the top of the
           stack to the resultat as long as the priority of the top of the
           stack is less than or equal to the priority of the element, push
           the element on the stack
pop the stack and add the top of stack to the
result until the stack is empty
```

As you can see, it is crucial in the algorithm that there are assigned the right priorities for the individual tokens. It is the priorities that determine when to move from the stack to the result, which is just a list. The two important points in the algorithm are 6 and 7. If you get to an operator – for example a multiplication – you must first move everything on the stack with a better priority than multiplication to the result list. It will be numbers, variables and functions, and then the multiplication operator is put on the stack.

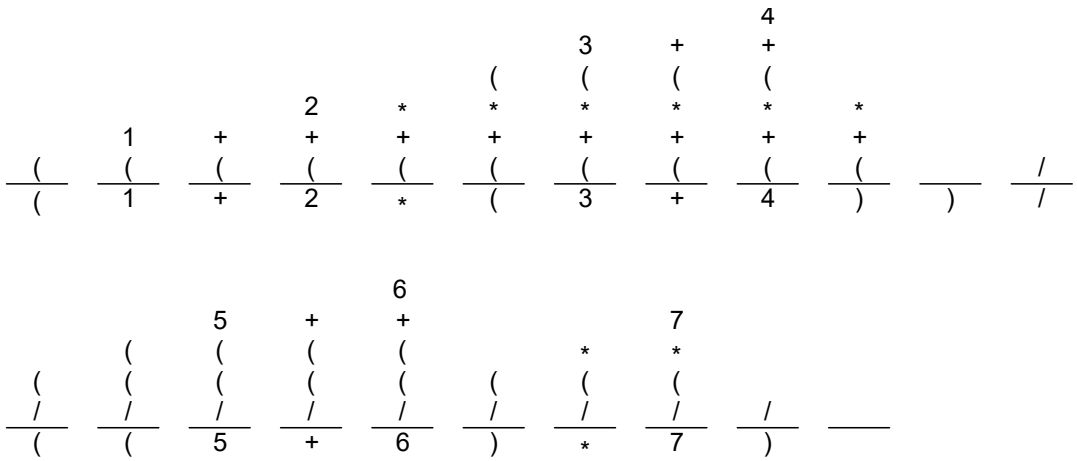
So, if the expression's tokens are converted to postfix form, it is simple to implement method *getValue()* in the class *Expression*.

If you look at the expression

$$(1 + 2 * (3 + 4))/((5 + 6) * 7)$$

it can be converted to postfix form in the following manner:

```
1
1 2
1 2 3
1 2 3 4 +
1 2 3 4 + * +
1 2 3 4 + * + 5
1 2 3 4 + * + 5 6 +
1 2 3 4 + * + 5 6 + 7 *
1 2 3 4 + * + 5 6 + 7 * /
```



In this particular task, I will apply the following priorities:

- numbers, variables 0
- sign 1
- function 2
- multiplication, division 3
- addition, subtraction 4
- left parenthes 9
- right parenthes, comma 99

8.3 PROGRAMMING

As is apparent from the above, the program will consist of many classes corresponding to the token classes. They are all in the same file, named *Tokens* that contain a single *public* class. It has a single static method called *toToken()* which, based on a substring representing the next token, and a *Token* object for the last token converts the substring to a token. The class is used by the *Expression* class as part of the scanning.

The class *Expression* is as mentioned, a complex class and after the scanning the class has a list of *Token* objects. These objects must be parsed to test the syntax of the expression, and for this purpose must be written a number of methods corresponding to the syntax diagrams. This methods uses recursion which I will mention below.

After the parsing, the list of tokens must be converted to postfix form, but with the above algorithm it is relatively simple. You should note, that the conversion needs a stack, that is a collection class as explained above. The same applies to implements the method *getValue()*, that calculates the value of an expression.

RECURSION

To write the class *Expression* (the syntax checker) I needs recursion and therefore a few words about what it is.

A method in a class can be seen as an isolated code that performs a specific operation on basis of parameters and possible returns a value. The method's statements, the commands it executes, can be all possible statements and there are no limitations on what it can be. For example calling another method, and a method may thus especially also calls the method itself. If so, wee say that the method is recursive. As an example of a recursive method – and a method which does not concern the specific program – is shown a method that determines the factorial of n:

```
static long factorial(int n)
{
 if (n == 0 || n == 1) return 1;
 return n * factorial(n - 1);
}
```

At a first glance, recursive methods can be hard to figure out, but once you have been familiar with the principle, it is not particularly difficult. Above is the principle that if *n* is 0 or 1, you can directly determine the result. If *n* is greater than 1, one can determine the factorial of *n* as *n* times the factorial of *n-1*. You can think of it in this way, to determine the factorial of *n* it is reduced to determine the factorial of *n-1*, which is a smaller a problem than I started with: To determine the factorial of *n*. If I repeat above a sufficient number of times, I get finally to the simple case in which *n* is 0 or 1 and where I can directly determine the result.

The principle of a recursive method is that a problem may be divided into two problems: A simple problem which can readily be solved, and a simpler (smaller) problem of the same kind as the original.

Formally, the factorial of n is defined as follows:

$$n! = \begin{cases} 1 \ for \ n = 0 \text{ or } n = 1 \\ n \cdot (n-1)! \ for \ n > 1 \end{cases}$$

and then by a recursive definition, and in such situations, recursion is often a good solution. In this case, the method *factorial()* simply just a rewrite of the mathematical formula to a method in Java.

It is clear that in this case the method could be written iteratively by means of a simple loop, and this solution would even be preferable, but in other situations, recursion is a good solution to provide a simple solutions and even a code which is easier to read and understand than an equivalent iterative solution. There is reason to always be aware of recursive methods, as each recursive call creates an activation block on the program stack. There is therefore a danger that the recursive methods use the whole stack, with the result that the program will crash.

THE VIEW

This leaves the program's view, which simply consists of a single class named *MainView*. The class contains nothing new and should not be discussed further here.

The program *Calc* has in contrast to the programs that I have shown so far in this series of books on software development, focus on algorithms, and thus how to solve a relatively complex problem using a program. Rather than show the code here (and there are over 1200 lines of code) should you study the code and its algorithms and here especially

- how the scan is done
- how the expression is parsed
- how an expression is converted to postfix
- how an expression evaluated

It happens all in class *Expression*, but the code is carefully documented.

APPENDIX A

Applications must be tested before they can be used, and for example you can try out the program, and check whether the program gives the correct result. It will always be a part of a test, since it is here that one realizes where the program to the user works as it should. You must remember to test how the application behaves when you enter something illegal or the program use arguments that are not legal, and such, and it is also important to test what happens when the window is resized. Besides, it is important that such user tests are performed by others than who wrote the program when there is a great risk that the programmer because of complicity overlook anything: You never know what real users can find on!

Sometimes you can not do anything other than what I have mentioned above, but in other contexts it is necessary to test the code on a lower level, where the programmer tests the code. I will mention three important ways:

1. Comment the code.
2. Debug the code.
3. Unit test the code.

and I will as an example use the program *CurrrencyProgram*.

COMMENT THE CODE

All three methods are important, but each with their opportunities. I have previously mentioned that to write comment's in the code as an effective (and often overlooked) means to ensure the quality of the finished program. In principle, it is trivial, but you should not underestimate the process. For the sake of future maintenance is good documentation clearly important, but more important is the process of writing the documentation, if it does otherwise performed conscientiously.

I think you have to document the following:

- In front of each class write a comment that tells of the class's purpose and its genesis, but generally you should write everything that you believe that that a person that in the future should maintain the code, needs to know. The comment is updated by any subsequent substantial change of the class, and is described as a Java comment, so that it becomes part of a complete class documentation.
- All variables and other data definitions are documentated with a short comment for what a variable is use for. Here I not use Java comments, unless it is a public static variabel or field.

- All public methods are documented with Java comments, so they are included in a part of the final class documentation. All parameters, return values and exceptions are documented, but in general there should also be a description of the method's purpose and including pre conditions of the parameters and possible side effects. For simple *get* methods, it is typically sufficient to comment the return value.
    - All private and protected methods are in principle documented in the same way, but here I usually do not use Java comments. The documentation of these methods should not be included in the final class documentation.
    - Finally, there are algorithms in which it may be necessary to comment on the details of the code. Generally I leave those comments to be part of the comment in front of the method, and to endeavor to avoid comments inside the body of the method, but if it is necessary to explain how the algorithm work, of course not refrain from such comments.

As mentioned, the process is important because as a part of the job thinking through why you now have written the code that you have. I would not say that in this way, all errors are found, but you will find an incredible number of inconveniences and places where the solution is ineffective.

It is an extensive work documenting code, but the work is well spent!

It is not all code, I document with comments and it is typically the controller and model classes. However, it is rare to documents classes in the user interface and at least only to a limited extent. I feel simply not that it makes any special dividends, whether as future documentation or detection of inexpediencies. The reason is probably that the development of the user interface is largely reuse of code from other programs.

The project *CurrencyProgram* is documented according to those recommendations.

JAVA DOC

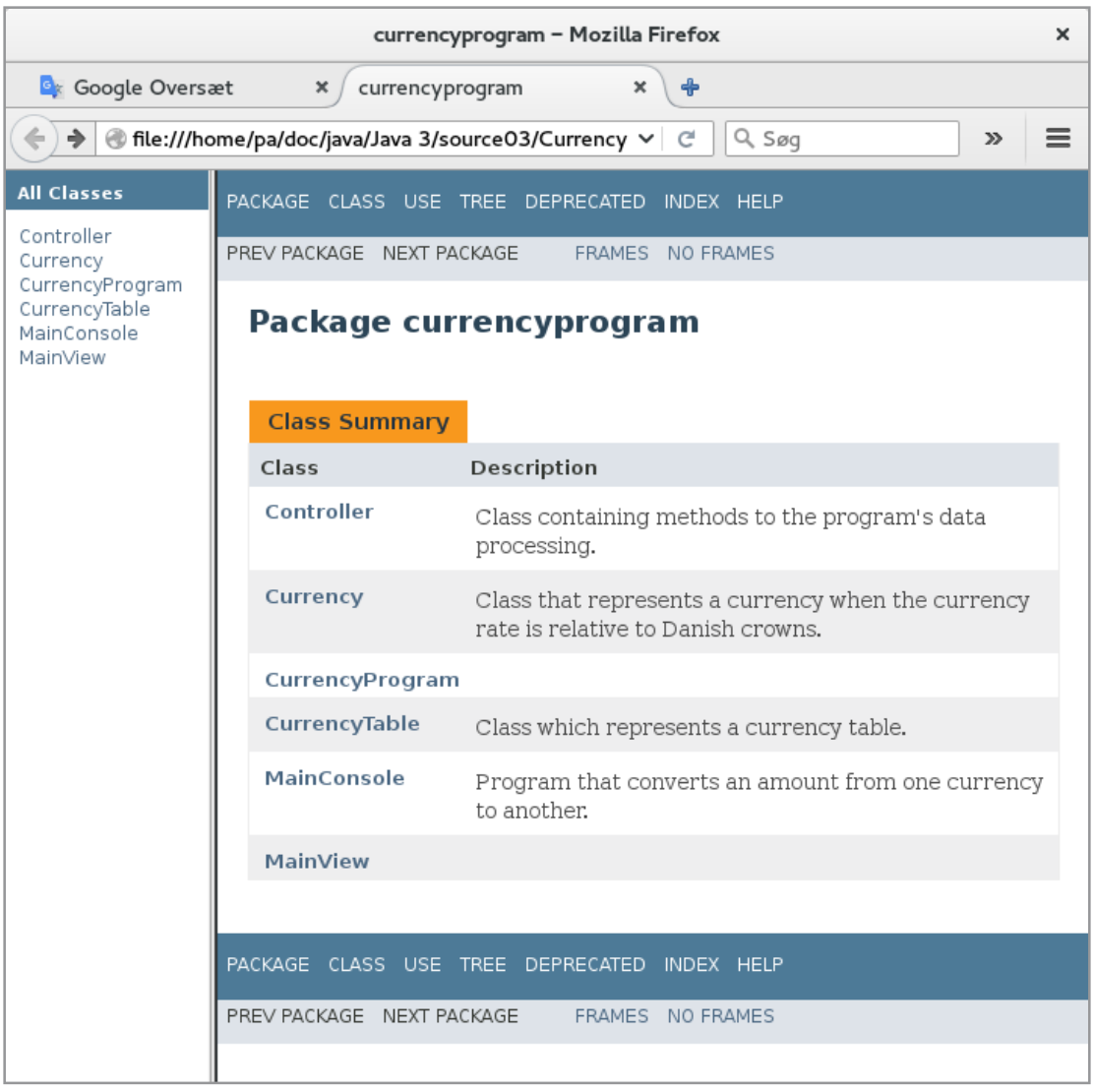
As mentioned above, you should widely use Java comments, and I also discussed how to get NetBeans to insert a skeleton in front of a method. As shown many times there is a special syntax for inserting comments from the parameters, return values, etc.. and there are actually some more, where the most important are:

- *@author*, the author of the class
- *{@code}* Display text in code font without interpreting the test as HTML
- *@exception* Adds a *Throws* heading to the generated documentation, with the classname and a description.

- *@{link}* Inserts an inline link with the visible text label that points to the documentation for the specified package, class, or member name of a referenced class.
- *@param* Adds a parameter with the specified parameter name followed by the specified description of the parameter.
- *@return* Adds a *Returns* section with a description.
- *@see* Adds a *See Also* heading with a link or text entry that points to a reference.

The idea is that by using a tool these comments can generate a complete documentation as an HTML document, and you can even write your own HTML in the docmentation.

If you have a project like *Currency* and in NetBeans right clicks on the project name, you get a menu item *Generate Javadoc*, which generates the documentation. NetBeans will then open a browser with the the result, which you can save. Below is shown the documentation of the current project:



DEBUG THE CODE

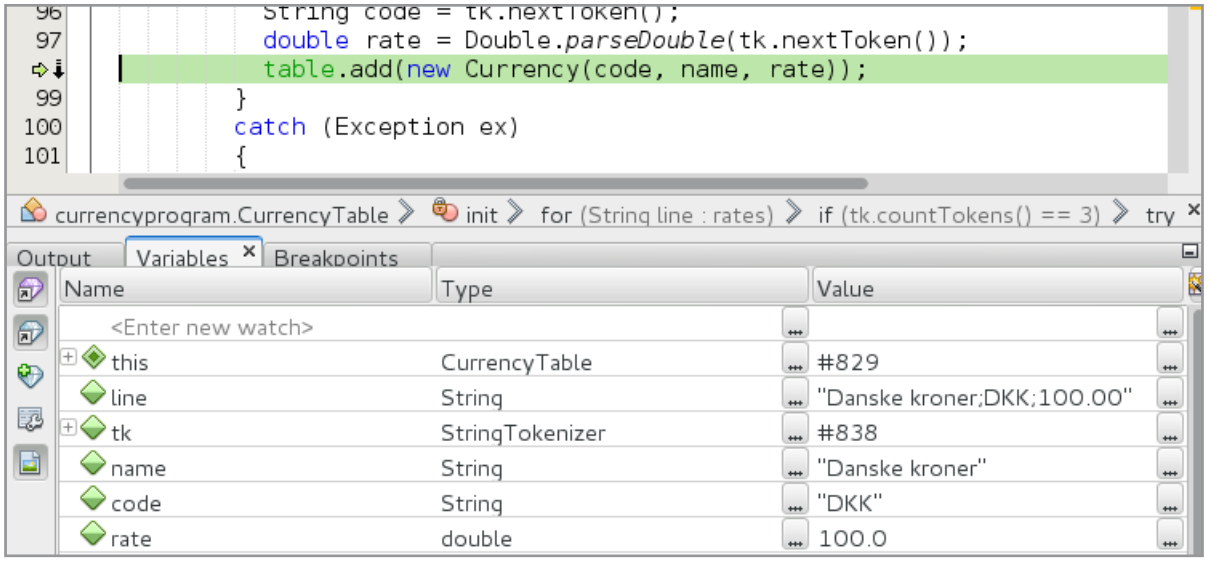
Now it may not be entirely correct to call debug the code for part of the test, but NetBeans has a good debugger, and since I have not previously referred to it, it must have a few words at this location.

When you want to test a program, it will often fail, and perhaps you get a wrong result, or the program crashes. The task then is to find the error and correct it, and here the debugger can help. You can set a breakpoint, which means that the excution of the program stops when the program reaches that point in the code, and you can then see the value of variables and check whether they have the right value. You can also step forward in the code statement by statement and constantly monitor what happens with variables. The debugger is a highly efficient tool to find where a program failed, but in general the debugger is used to analyze the code.

As an example, one might think that the program *CurrencyProgram* fails, when you create a new *CurrencyTable* object (which is a singleton, and there is a possibility of an error when the object is instantiated), and the task is then to find out where it goes wrong. As an example, one could then set a breakpoint in the method *init()*:

```
82
83      // Builds a currency table from the following array of strings.
84      // In case of an error where a string can not be parsed into a currency
85      // an error message and the program is terminated.
86      private void init()
87      {
88        for (String line : rates)
89        {
90          StringTokenizer tk = new StringTokenizer(line, ";");
91          if (tk.countTokens() == 3)
92          {
93            try
94            {
95              String name = tk.nextToken();
96              String code = tk.nextToken();
97              double rate = Double.parseDouble(tk.nextToken());
98              table.add(new Currency(code, name, rate));
99            }
100           catch (Exception ex)
101           {
102             System.out.println(ex.getMessage() + "\n" + line);
103           }
104         }
105         else System.out.println("Error: " + line);
106       }
107     }
```

You set the breakpoint by clicking with the mouse next to the desired line (here line 88). From the menu you can then start the debugger by selecting *Debug* | *Debug Project*. When *init()* is performed, the program will stop, where there is set a breakpoint, and you can then step through the code line by line (by using F8). That way you can see where the program possibly crashes. You can also put a watch on variables (by right-clicking in the debug window), and you can see the variables values and what happens to them:



The debugger has many other opportunities to analyze the code. For example you can by pressing F7 jump into the code for a method. You should examine the Debug menu to get an idea of what is possible. It is worthwhile to use some time learning the debugger to know, since it is a highly effective tool for troubleshooting. Probably it is not directly a testing tool, but a good tool for finding the errors detected during the testing.

UNIT TEST

Unit test is carried out with a tool, that is integrated into NetBeans. Unlike the testing of the program, where you test how the finished program behaves, and whether it meets all requirements, unit test is a systematic way to test individual classes and methods, and therefore it is something that must be done by the programmer as part of the development. It is not everything that you can unit test, and it is best suited for testing controller and model classes, and specifically unit test is effective test of classes in libraries.

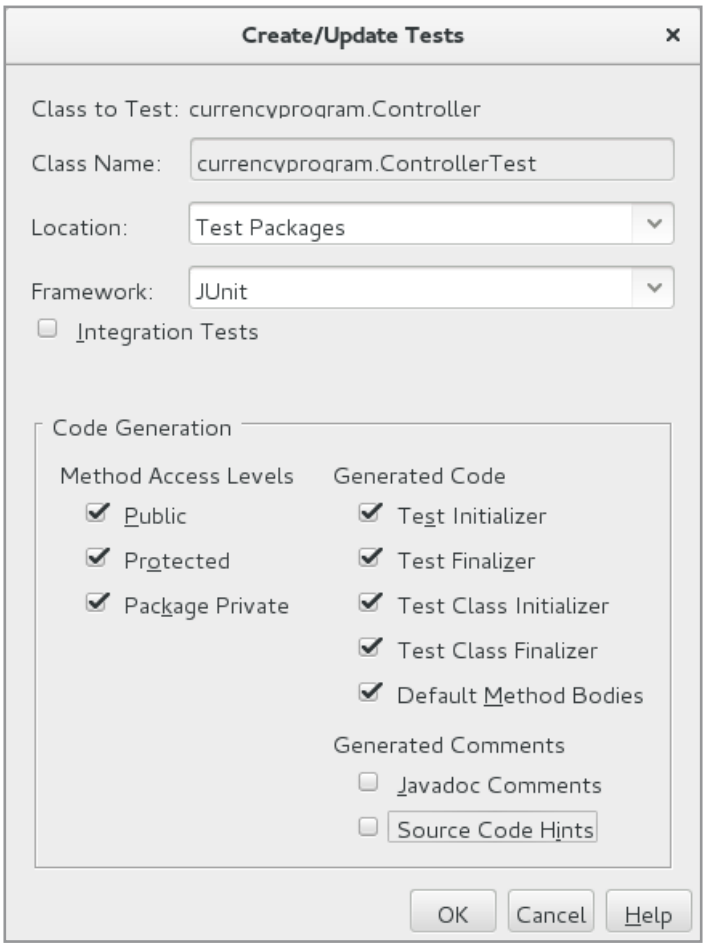
I will test the class *Controller*, which has three methods:

```
package currencyprogram;

import java.util.*;
import java.io.*;

/**
 * Class containing methods to the program's data processing.
 */
public class Controller
{
 /**
  * Converter an amount from one currency to another currency.
  * The conversion is done primarily through the next calculation method, and this
  * method should primarily validate that amount is a legal number.
  * @param amount The amount to be converted
  * @param from The currency to be converted from
  * @param to The currency to be converted to
  * @return The result of the conversion
  * @throws Exception The amount can not be parsed or an currency objects is null
  */
 public double calculate(String amount, Currency from, Currency to)
  throws Exception
 {
  …
 }
```

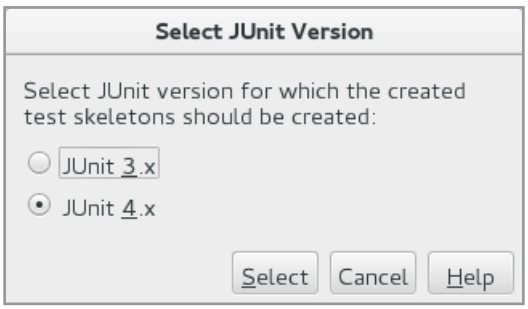
```
/**
 * Converter an amount from one currency to another currency.
 * @param amount The amount to be converted
 * @param from The currency to be converted from
 * @param to The currency to be converted to
 * @return The result of the conversion
 * @throws Exception If one of the two currency objects are null
 */
public double calculate(double amount, Currency from, Currency to)
 throws Exception
{
 …
}

/**
 * Updating the currency table from a semicolon delimited text file.
 * Each line consists of three fields separated by semicolons and in the
 * following order:
 *  name;code;rate
 * @param file The file from which to read currencies
 * @return A list containing all lines that could not be converted into currency
 */
```

```
  public ArrayList<String> update(File file)
  {
    …
  }
}
```

First, I creates a test class. You do this by right-clicking on the class name *Controller.java* in the *Projects* tab, and here should you choose *Tools* | *Create / Update Tests*



I kept all the settings as they are. When you click OK, you may receive a dialog box:



and if so, choose JUnit 4. Then NetBeans creates a test class, as shown below. The test class is called *ControllerTest*, and besides a default constructor is created four empty methods as well as three test methods. There are generally created a test method for each non-private method in the class. When the test class is carried out, this means that all of the test methods are executed. The four empty methods are used to add code that you want performed, respectively before and after the test, and two methods to be performed before and after the individual test methods. In this case I will not add anything and the methods could easily be deleted.

```
package currencyprogram;

import java.io.File;
import java.util.ArrayList;
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;
public class ControllerTest {

 public ControllerTest() {
 }

 @BeforeClass
 public static void setUpClass() {
 }

 @AfterClass
 public static void tearDownClass() {
 }

 @Before
 public void setUp() {
 }

 @After
 public void tearDown() {
 }
 @Test
 public void testCalculate_3args_1() throws Exception {
  System.out.println("calculate");
  String amount = "";
  Currency from = null;
```

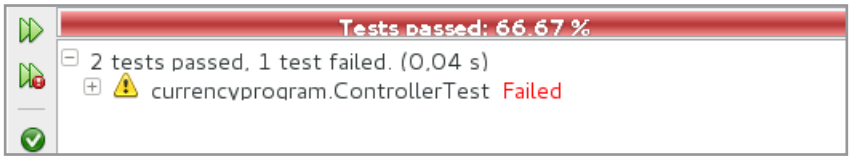
```
 Currency to = null;
 Controller instance = new Controller();
 double expResult = 0.0;
 double result = instance.calculate(amount, from, to);
 assertEquals(expResult, result, 0.0);
 fail("The test case is a prototype.");
}

@Test
public void testCalculate_3args_2() throws Exception {
 System.out.println("calculate");
 double amount = 0.0;
 Currency from = null;
 Currency to = null;
 Controller instance = new Controller();
 double expResult = 0.0;
 double result = instance.calculate(amount, from, to);
 assertEquals(expResult, result, 0.0);
 fail("The test case is a prototype.");
}
```

```
@Test
public void testUpdate() {
 System.out.println("update");
 File file = null;
 Controller instance = new Controller();
 ArrayList<String> expResult = null;
 ArrayList<String> result = instance.update(file);
 assertEquals(expResult, result);
 fail("The test case is a prototype.");
}
```

You must specifically noting how the test methods are named, and how to solve the problem that two methods has the same name. The auto-generated test methods can usually not be used directly for anything, but you can notice that the methods includes usual Java statements, and the work of writing unit tests is to write code to test methods. As a start, I have changed the middle test method to the following while I have comment out the code in the other two:

```
@Test
public void testCalculate_3args_2() throws Exception {
 System.out.println("calculate(double amount, Currency from, Currency to)");
 Currency c1 = new Currency("DKK", "Danish crowns", 100);
 Currency c2 = new Currency("EUR", "Euro", 750);
 Currency c3 = new Currency("USD", "US dollar", 600);
 Controller instance = new Controller();
 assertEquals(instance.calculate(1000, c2, c1), 7500, 0.0001);
 assertEquals(instance.calculate(1000, c1, c2), 133.33, 0.01);
 assertEquals(instance.calculate(1000, c2, c3), 1200, 0.0001);
 assertEquals(instance.calculate(1000, c3, c2), 800, 0.0001);
}
```

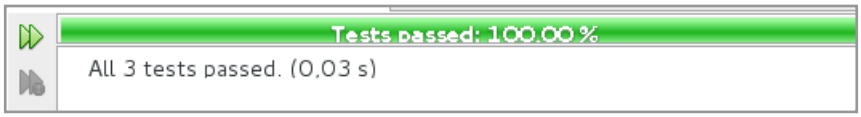
Wee say that the method has four test cases. Each test case performs the method *assertEquals()* with two currency objects as parameters. Every time the method's return value is tested whether it has a certain value and because the type is *double*, you must with the last parameter set the maximum deviation. If a test case fails, the test is interrupted with an error message. That is to accept the test all tests cases must be performed without error. That's what the unit test is about, and the only thing to learn is which test cases it is possible to write. To perform the test, right-click the test class under the *Projects* tab and choose *Test File*. The result says where the test is performed correctly:



In this case, the test failed. The reason is that the third test case fails, since – erroneously – there is an incorrect expected value. I change it to

```
assertEquals(instance.calculate(1000, v2, v3), 1250, 0.0001);
```

and the test is performed correct:



The value of such tests is of course determined by the number of test cases. Below is the code for the first test method:

```
@Test
public void testCalculate_3args_1() throws Exception {
 System.out.println("calculate(String amount, Currency from, Currency to)");
 Currency c1 = new Currency("DKK", "Danish crowns", 100);
 Currency c2 = new Currency("EUR", "Euro", 750);
 Currency c3 = new Currency("USD", "US dollar", 600);
 Controller instance = new Controller();
 assertTrue(Math.abs(instance.calculate("1000", c2, c1) - 7500) < 0.01);
 assertTrue(Math.abs(instance.calculate("1000", c1, c2) - 133.33) < 0.01);
 assertTrue(Math.abs(instance.calculate("1000", c2, c3) - 1250) < 0.01);
 assertTrue(Math.abs(instance.calculate("1000", c3, c2) - 800) < 0.01);
}
```

It looks like the previous test method, and the difference between the two methods is also only the type of the first parameter. Each test case is written this time in a different manner by means of *assertTrue()*. There are no special reasons for it in addition to showing the syntax.

You can also add your own test methods, and the following method is used to test whether you get an exception if the *calculate()* method is performed with a *Currency* that is *null*:

```
@Test(expected=Exception.class)
public void testNullValuta() throws Exception {
 System.out.println("calculate with arguments that is null");
 Currency c1 = new Currency("DKK", "Danish crowns", 100);
 Controller instance = new Controller();
```

```
 assertEquals(instance.calculate(1000, c1, null), 7500, 0.0001);
 assertEquals(instance.calculate(1000, null, c1), 133.33, 0.01);
}
```

You should note, how to specify that a test case may raise an exception.

Next I adds a test class for the class *CurrencyTable*. This is done in exactly the same manner as above, and the results are as follows:

```
package currencyprogram;

import java.util.Iterator;
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;
public class CurrencyTableTest {

 public CurrencyTableTest() {
 }
```

```
@BeforeClass
public static void setUpClass() {
}

@AfterClass
public static void tearDownClass() {
}

@Before
public void setUp() {
}

@After
public void tearDown() {
}
@Test
public void testGetInstance() {
 System.out.println("getInstance");
 CurrencyTable expResult = null;
 CurrencyTable result = CurrencyTable.getInstance();
 assertEquals(expResult, result);
 fail("The test case is a prototype.");
}

@Test
public void testGetCurrency() throws Exception {
 System.out.println("getCurrency");
 String code = "";
 CurrencyTable instance = null;
 Currency expResult = null;
 Currency result = instance.getCurrency(code);
 assertEquals(expResult, result);
 fail("The test case is a prototype.");
}

@Test
public void testIterator() {
 System.out.println("iterator");
 CurrencyTable instance = null;
 Iterator<Currency> expResult = null;
 Iterator<Currency> result = instance.iterator();
 assertEquals(expResult, result);
 fail("The test case is a prototype.");
}
@Test
public void testUpdate() {
 System.out.println("update");
```

```
   Currency currency = null;
   CurrencyTable instance = null;
   boolean expResult = false;
   boolean result = instance.update(currency);
   assertEquals(expResult, result);
   fail("The test case is a prototype.");
  }
}
```

This time there are four test methods. The first test the static method *getInstance()*, which simply returns a reference to the singleton. It makes no sense, and I will delete it. So you can simply delete the test methods which you do not wish to make use of. I would also delete the test method that tests the iterator. I will now's writing the code for the last two test methods:

```
@Test
public void testGetCurrency() throws Exception {
 System.out.println("getCurrency(String code)");
 assertNotNull(CurrencyTable.getInstance().getCurrency("DKK"));
 assertNotNull(CurrencyTable.getInstance().getCurrency("EUR"));
}

@Test
public void testUpdate() {
 System.out.println("update(Currency currency)");
 try
 {
  int count1 = 0;
  for (Currency c : CurrencyTable.getInstance()) ++count1;
  Currency c1 = new Currency("EUR", "European euro", 800);
  Currency c2 = new Currency("TST", "Test currency", 1000);
  CurrencyTable.getInstance().update(c1);
  CurrencyTable.getInstance().update(c2);
  Currency c3 = CurrencyTable.getInstance().getCurrency("EUR");
  Currency c4 = CurrencyTable.getInstance().getCurrency("TST");
  int count2 = 0;
  for (Currency c : CurrencyTable.getInstance()) ++count2;
  System.out.println(c3.getName());
  System.out.println(c4.getName());
  assertEquals(c3.getRate(), 800, 0.0001);
  assertEquals(c4.getRate(), 1000, 0.0001);
  assertEquals(count2 - count1, 1);
 }
 catch (Exception ex)
 {
  System.out.println(ex);
 }
}
```

There is not much to explain, but note that the first use *assertNotNull()* to test that an object is not *null*. Note also that this time you should not create an instance of the *CurrencyTable*, as the class is written as a singleton. Finally, note the last test method, and that a test method can include all the Java statements that may be needed.

Back there is a test method in the test class *ControllerTest*, but I will not write the code, but note that it is simple to write a file with test data and test if the method is working properly.

The strength of a unit test depends as mentioned by the number of test cases. It can be a lot of work to write test classes for unit testing, but classes must be tested, and the real value of the unit test is that if first written individual test methods you can repeat the test all the times that may be required, and it is, in principle, every time the class is modified.

As a final note regarding unit test NetBeans has a menu *Run* | *Test Project*, and if you choose that, all test classes are performed.